

Mike McQuaid
Foreword by Scott Chacon



Git

IN PRACTICE

INCLUDES 66 TECHNIQUES



Git in Practice

by Mike McQuaid

Chapters 3, 5, and 13
ISBN 9781617291975

Copyright 2015 Manning Publications
To pre-order or learn more about this book go to www.manning.com/mcquaid/

brief contents

PART 1	INTRODUCTION TO GIT.....	1
1	■ Local Git	3
2	■ Remote Git	24
PART 2	GIT ESSENTIALS.....	51
3	■ Filesystem interactions	53
4	■ History visualization	68
5	■ Advanced branching	84
6	■ Rewriting history and disaster recovery	104
PART 3	ADVANCED GIT	127
7	■ Personalizing Git	129
8	■ Vendoring dependencies as submodules	141
9	■ Working with Subversion	151
10	■ GitHub pull requests	163
11	■ Hosting a repository	174

PART 4	Git BEST PRACTICES	185
12	■ Creating a clean history	187
13	■ Merging vs. rebasing	196
14	■ Recommended team workflows	206

Filesystem interactions

This chapter covers

- Renaming, moving, and removing versioned files or directories
- Telling Git to ignore certain files or changes
- Deleting all untracked or ignored files or directories
- Resetting all files to their previously committed state
- Temporarily stashing and reapplying changes to files

When working with a project in Git, you'll sometimes want to move, delete, change, and/or ignore certain files in your working directory. You could mentally keep track of the state of important files and changes, but this isn't a sustainable approach. Instead, Git provides commands for performing filesystem operations for you.

Understanding the Git filesystem commands will allow you to quickly perform these operations rather than being slowed down by Git's interactions. Let's start with the most basic file operations: renaming or moving a file.

Technique 17 Renaming or moving a file: *git mv*

Git keeps track of changes to files in the working directory of a repository by their name. When you move or rename a file, Git doesn't see that a file was moved; it sees that there's a file with a new filename, and the file with the old filename was deleted (even if the contents remain the same). As a result, renaming or moving a file in Git is essentially the same operation; both tell Git to look for an existing file in a new location. This may happen if you're working with tools (such as IDEs) that move files for you and aren't aware of Git (and so don't give Git the correct move instruction).

Sometimes you'll still need to manually rename or move files in your Git repository, and want to preserve the history of the files after the rename or move operation. As you learned in technique 4, readable history is one of the key benefits of a version control system, so it's important to avoid losing it whenever possible. If a file has had 100 small changes made to it with good commit messages, it would be a shame to undo all that work just by renaming or moving a file.

Problem

In your Git working directory, you wish to rename a previously committed file named `GitInPractice.asciidoc` to `01-IntroducingGitInPractice.asciidoc` and commit the newly renamed file.

Solution

- 1 Change to the directory containing your repository: for example, `cd /Users/mike/GitInPracticeRedux/`.
- 2 Run `git mv GitInPractice.asciidoc 01-IntroducingGitInPractice.asciidoc`. There will be no output.
- 3 Run `git commit --message 'Rename book file to first part file.'`. The output should resemble the following.

Listing 3.1 Output: renamed commit

```
# git commit --message 'Rename book file to first part file.'
[master c6eed66] Rename book file to first part file.
1 file changed, 0 insertions(+), 0 deletions(-)
rename GitInPractice.asciidoc =>
    01-IntroducingGitInPractice.asciidoc (100%)
```

Commit message → [master c6eed66] Rename book file to first part file.

No insertions/deletions ← 0 insertions(+), 0 deletions(-)

Old filename changed to new filename ← GitInPractice.asciidoc => 01-IntroducingGitInPractice.asciidoc

You've renamed `GitInPractice.asciidoc` to `01-IntroducingGitInPractice.asciidoc` and committed it.

Discussion

Moving and renaming files in version control systems rather than deleting and re-creating them is done to preserve their history. For example, when a file has been moved into a new directory, you'll still be interested in the previous versions of the file before it was moved. In Git's case, it will try to auto-detect renames or moves on `git`

`add` or `git commit`; if a file is deleted and a new file is created, and those files have a majority of lines in common, Git will automatically detect that the file was moved and `git mv` isn't necessary. Despite this handy feature, it's good practice to use `git mv` so you don't need to wait for a `git add` or `git commit` for Git to be aware of the move and so you have consistent behavior across different versions of Git (which may have differing move auto-detection behavior).

After running `git mv`, the move or rename will be added to Git's index staging area, which, if you remember from technique 2, means the change has been staged for inclusion in the next commit.

It's also possible to rename files or directories and move files or directories into other directories in the same Git repository using the `git mv` command and the same syntax as earlier. If you want to move files into or out of a repository, you must use a different, non-Git command (such as a Unix `mv` command), because Git doesn't handle moving files between different repositories with `git mv`.

WHAT IF THE NEW FILENAME ALREADY EXISTS? If the filename you move to already exists, you'll need to use the `git mv -f` (or `--force`) option to request that Git overwrite whatever file is at the destination. If the destination file hasn't already been added or committed to Git, then it won't be possible to retrieve the contents if you erroneously asked Git to overwrite it.

Technique 18 *Removing a file: git rm*

Like moving and renaming files, removing files from version control systems requires not just performing the filesystem operation as usual, but also notifying Git and committing the file. In almost any version-controlled project, you'll at some point want to remove some files, so it's essential to know how to do so. Removing version-controlled files is also safer than removing non-version-controlled files because even after removal, the files still exist in the history.

Sometimes tools that don't interact with Git may remove files for you and require you to manually indicate to Git that you wish these files to be removed. For testing purposes, let's create and commit a temporary file to be removed:

- 1 Change to the directory containing your repository; for example,
`cd /Users/mike/GitInPracticeRedux/`.
- 2 Run `echo Git Sandwich > GitInPracticeReviews.tmp`. This creates a new file named `GitInPracticeReviews.tmp` with the contents "Git Sandwich".
- 3 Run `git add GitInPracticeReviews.tmp`.
- 4 Run `git commit --message 'Add review temporary file.'`

Note that if `git add` fails, you may have `*.tmp` in a `.gitignore` file somewhere (introduced in technique 21). In this case, add it using `git add --force GitInPracticeReviews.tmp`.

Problem

You wish to remove a previously committed file named `GitInPracticeReviews.tmp` in your Git working directory and commit the removed file.

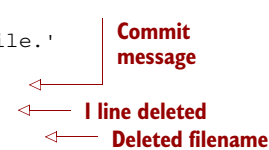
Solution

- 1 Change to the directory containing your repository; for example,
`cd /Users/mike/GitInPracticeRedux/.`
- 2 Run `git rm GitInPracticeReviews.tmp`.
- 3 Run `git commit --message 'Remove unfavourable review file.'`. The output should resemble the following.

Listing 3.2 Output: removed commit

```
# git rm GitInPracticeReviews.tmp
rm 'GitInPracticeReviews.tmp'

# git commit --message 'Remove unfavourable review file.'
[master 06b5eb5] Remove unfavourable review file.
1 file changed, 1 deletion(-)
delete mode 100644 GitInPracticeReviews.tmp
```



You've removed `GitInPracticeReviews.tmp` and committed it.

Discussion

Git only interacts with the Git repository when you explicitly give it commands, which is why when you remove a file, Git doesn't automatically run a `git rm` command. The `git rm` command is indicating to Git not just that you wish for a file to be removed, but also (like `git mv`) that this removal should be part of the next commit.

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.

WHAT IF A FILE HAS UNCOMMITTED CHANGES? If a file has uncommitted changes but you still wish to remove it, you need to use the `git rm -f` (or `--force`) option to indicate to Git that you want to remove it before committing the changes.

Technique 19 Resetting files to the last commit: `git reset`

There are times when you've made changes to files in the working directory but you don't want to commit these changes. Perhaps you added debugging statements to files

and have now committed a fix, so you want to reset all the files that haven't been committed to their last committed state (on the current branch).

Problem

You wish to reset the state of all the files in your working directory to their last committed state.

Solution

- 1 Change to the directory containing your repository: for example,
`cd /Users/mike/GitInPracticeRedux/`.
- 2 Run `echo EXTRA >> 01-IntroducingGitInPractice.asciidoc` to append "EXTRA" to the end of `01-IntroducingGitInPractice.asciidoc`.
- 3 Run `git reset --hard`. The output should resemble the following.

Listing 3.3 Output: hard reset

```
# git reset --hard
```

```
HEAD is now at 06b5eb5 Remove unfavourable review file.
```

Reset
commit

You've reset the Git working directory to the last committed state.

Discussion

The `--hard` argument signals to `git reset` that you want it to reset both the 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 `git add`, but `git reset --hard` undoes `git add` and all file modifications.

`git reset` will be used to perform more operations (including those that alter history) later, in technique 42.

DANGERS OF USING GIT RESET --HARD Take care when you use `git reset --hard`; it will immediately and without prompting remove all uncommitted changes to any file in your working directory. This is probably the command that has caused me more regret than any other; I've typed it accidentally and removed work I hadn't intended to. Remember that in section 1.1 you learned that it's very hard to lose work with Git? If you have uncommitted work, this is one of the easiest ways to lose it! A safer option may be to use Git's stash functionality instead.

Technique 20 *Deleting untracked files: git clean*

When working in a Git repository, some tools may output undesirable files into your working directory. Some text editors may use temporary files, operating systems may write thumbnail cache files, or programs may write crash dumps. Alternatively, sometimes there may be files that are desirable, but you don't wish to commit them

to your version control system; instead you want to remove them to build clean versions (although this is generally better handled by *ignoring* these files, as shown in technique 21).

When you wish to remove these files, you could remove them manually. But it's easier to ask Git to do so, because it already knows which files in the working directory are versioned and which are *untracked*.

You can view the files that are currently tracked by running `git ls-files`. This currently only shows `01-IntroducingGitInPractice.asciidoc`, because that is the only file that has been added to the Git repository. You can run `git ls-files --others` (or `-o`) to show the currently untracked files (there should be none).

For testing purposes, let's create a temporary file to be removed:

- 1 Change to the directory containing your repository; for example,
`cd /Users/mike/GitInPracticeRedux/`.
- 2 Run `echo Needs more cowbell > GitInPracticeIdeas.tmp`. This creates a new file named `GitInPracticeIdeas.tmp` with the contents "Needs more cowbell".

Problem


You wish to remove an untracked file named `GitInPracticeIdeas.tmp` from a Git working directory.

Solution

- 1 Change to the directory containing your repository; for example,
`cd /Users/mike/GitInPracticeRedux/`.
- 2 Run `git clean --force`. The output should resemble the following.

Listing 3.4 Output: force-cleaned files

```
# git clean --force
Removing GitInPracticeIdeas.tmp
```



Removed
file

You've removed `GitInPracticeIdeas.tmp` from the Git working directory.

Discussion

`git clean` requires the `--force` argument because this command is potentially dangerous; with a single command, you can remove many, many files very quickly. Remember that in section 1.1, you learned that accidentally losing any file or change committed to the Git system is nearly impossible. This is the opposite situation; `git clean` will happily remove thousands of files very quickly, and they can't be easily recovered (unless you backed them up through another mechanism).

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 21 Ignoring files: .gitignore

As discussed in technique 20, sometimes working directories contain files that are *untracked* by Git, and you don’t want to add them to the repository. Sometimes these files are one-off occurrences; you accidentally copy a file to the wrong directory and want to delete it. More often, they’re the product of software (such as the software stored in the version control system or some part of your operating system) putting files into the working directory of your version control system.

You could `git clean` these files each time, but that would rapidly become tedious. Instead, you can tell Git to ignore them so it never complains about these files being untracked and you don’t accidentally add them to commits.

Problem

You wish to ignore all files with the extension `.tmp` in a Git repository.

Solution

- 1 Change to the directory containing your repository: for example,
`cd /Users/mike/GitInPracticeRedux/`.
- 2 Run `echo *.tmp > .gitignore`. This creates a new file named `.gitignore` with the contents “`*.tmp`”.
- 3 Run `git add .gitignore` to add `.gitignore` to the index staging area for the next commit. There will be no output.
- 4 Run `git commit --message='Ignore .tmp files.'`. The output should resemble the following.

Listing 3.5 Output: ignore file commit

```
# git commit --message='Ignore .tmp files.'
```

```
[master 0b4087c] Ignore .tmp files.
1 file changed, 1 insertion(+)
create mode 100644 .gitignore
```

Commit message

1 line deleted

Created filename

You’ve added a `.gitignore` file with instructions to ignore all `.tmp` files in the Git working directory.

Discussion

Each line of a `.gitignore` file matches files with a pattern. For example, you can add comments by starting a line with a `#` character or negate patterns by starting a line with a `!` character. Read more about the pattern syntax in `git help gitignore`.

A good and widely held principle for version control systems is to avoid committing *output files* to a version control repository. Output files are those that are created from input files that are stored in the version control repository.

For example, you may have a `hello.c` file that is compiled into a `hello.o` object file. The `hello.c` *input file* should be committed to the version control system, but the `hello.o` *output file* should not.

Committing `.gitignore` to the Git repository makes it easy to build up lists of expected output files so they can be shared between all the users of a repository and not accidentally committed. GitHub also provides a useful collection of gitignore files at <https://github.com/github/gitignore>.

Let's try to add an ignored file:

- 1 Change to the directory containing your repository; for example,
`cd /Users/mike/GitInPracticeRedux/`.
- 2 Run `touch GitInPracticeGoodIdeas.tmp`. This creates a new, empty file named `GitInPracticeGoodIdeas.tmp`.
- 3 Run `git add GitInPracticeGoodIdeas.tmp`. The output should resemble the following.

Listing 3.6 Output: trying to add an ignored file

```
# git add GitInPracticeGoodIdeas.tmp
```

The following paths are ignored by one of your `.gitignore` files:

```
GitInPracticeGoodIdeas.tmp
```

Use `-f` if you really want to add them.

```
fatal: no files added
```

← 1 Ignored file

← 2 Error message

1 `GitInPracticeGoodIdeas.tmp` wasn't added, because its addition would contradict your `.gitignore` rules.

2 was printed, because no files were added.

This interaction between `.gitignore` and `git add` is particularly useful when adding subdirectories of files and directories that may contain files that should to be ignored. `git add` won't add these files but will still successfully add all others that shouldn't be ignored.

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. For example, you may have a project in a Git repository that compiles input files (such as `.c` files) into output files (in this example, `.o` files) and wish to remove all these output files from the working directory to perform a new build from scratch.

Let's create some temporary files that can be removed:

- 1 Change to the directory containing your repository: for example,
`cd /Users/mike/GitInPracticeRedux/.`
- 2 Run `touch GitInPracticeFunnyJokes.tmp GitInPracticeWittyBanter.tmp`.

Problem

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

Solution

- 1 Change to the directory containing your repository: for example,
`cd /Users/mike/GitInPracticeRedux/.`
- 2 Run `git clean --force -X`. The output should resemble the following.

Listing 3.7 Output: force-cleaning ignored files

```
# git clean --force -X
```

```
Removing GitInPracticeFunnyJokes.tmp
```

← Removed file

```
Removing GitInPracticeWittyBanter.tmp
```

You've removed all ignored files from the Git working directory.

Discussion

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: git stash

There are times when you may find yourself working on a new commit and want to temporarily undo your current changes but redo them at a later point. Perhaps there was an urgent issue that means you need to quickly write some code and commit a fix. In this case, you could make a temporary branch and merge it in later, but this would add a commit to the history that may not be necessary. 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.

Problem

You wish to stash all your uncommitted changes for later retrieval.

Solution

- 1 Change to the directory containing your repository; for example, `cd /Users/mike/GitInPracticeRedux/`.
- 2 Run `echo EXTRA >> 01-IntroducingGitInPractice.asciidoc`.
- 3 Run `git stash save`. The output should resemble the following.

Listing 3.8 Output: stashing uncommitted changes

```
# git stash save

Saved working directory and index state WIP on master:
36640a5 Ignore .tmp files.
HEAD is now at 36640a5 Ignore .tmp files.           ← Current commit
```

You’ve stashed your uncommitted changes.

Discussion

`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. It’s possible to access this commit directly, but you should only do so through `git stash` to avoid confusion.

You can see all the stashes that have been made by running `git stash list`. The output will resemble the following.

Listing 3.9 List of stashes

```
stash@{0}: WIP on master: 36640a5 Ignore .tmp files.   ← Stashed commit
```

This shows the single stash that you made. You can access it using `ref stash@{0}`; for example, `git diff stash@{0}` will show you the difference between the working directory and the contents of that stash.

If you save another stash, it will become `stash@{0}` and the previous stash will become `stash@{1}`. This is because the stashes are stored on a *stack* structure. A stack structure is best thought of as being like a stack of plates. You add new plates on the top of the existing plates; and if you remove a single plate, you take it from the top. Similarly, when you run `git stash`, the new stash is added to the top (it becomes `stash@{0}`) and the previous stash is no longer at the top (it becomes `stash@{1}`).

DO YOU NEED TO USE GIT ADD BEFORE GIT STASH? No, `git add` is not needed. `git stash` stashes your changes regardless of whether they’ve been added to the index staging area by `git add`.

DOES GIT STASH WORK WITHOUT THE SAVE ARGUMENT? 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.

Technique 24 Reapplying stashed changes: *git stash pop*

When you've stashed your temporary changes and performed whatever operations required a clean working directory (perhaps you fixed and committed the urgent issue), you'll want to reapply the changes (because otherwise you could've just run `git reset --hard`). When you've checked out the correct branch again (which may differ from the original branch), you can request that the changes be taken from the stash and applied onto the working directory.

Problem

You wish to pop the changes from the last `git stash` save into the current working directory.

Solution

- 1 Change to the directory containing your repository; for example,
`cd /Users/mike/GitInPracticeRedux/.`
- 2 Run `git stash pop`. The output should resemble the following.

Listing 3.10 Output: reapplying stashed changes

```
# git stash pop
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working
#    directory)
#
#    modified:   01-IntroducingGitInPractice.asciidoc
#
no changes added to commit (use "git add" and/or "git commit -a")
Dropped refs/stash@{0} (f7e39e2590067510be1a540b073e74704395e881)
```

Stashed commit →

Current branch output

Begin status output

End status output

You've reapplied the changes from the last `git stash` save.

Discussion

When running `git stash pop`, the top stash on the stack (`stash@{0}`) is applied to the working directory and removed from the stack. If there's a second stash in the stack (`stash@{1}`), it's now at the top (it becomes `stash@{0}`). This means if you run `git stash pop` multiple times, it will keep working down the stack until no more stashes are found, at which point it will output `No stash found`.

If you wish to apply an item from the stack multiple times (perhaps on multiple branches), 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.

Technique 25 *Clearing stashed changes: git stash clear*

You may have stashed changes with the intent of popping them later, but then realize that you no longer wish to do so—the changes in the stack are now unnecessary, so you want to get rid of them all. You could do this by popping each change off the stack and then deleting it, but it would be handy to have a command that allows you to do this in a single step. Thankfully, `git stash clear` does just this.

Problem

You wish to clear all previously stashed changes.

Solution

- 1 Change to the directory containing your repository; for example,
`cd /Users/mike/GitInPracticeRedux/`.
- 2 Run `git stash clear`. There will be no output.

You've cleared all the previously stashed changes.

Discussion

Clearing the stash is done without a prompt and removes every previous item from the stash, so be careful when doing so. Cleared stashes can't be easily recovered. For this reason, once you learn about history rewriting in technique 42, I'd recommend making commits and rewriting them later rather than relying too much on `git stash`.

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. In these cases, you could ignore them yourself or stash them elsewhere, but it would be ideal to be able to tell Git to ignore these particular changes.

I've found myself in a situation in the past where I wanted to test a Rails configuration file change for a week or two while continuing to do my normal work. I didn't want to commit it because I didn't want it to apply to servers or my coworkers, but I did want to continue testing it while I made other commits rather than change to a particular branch each time.

Problem

You wish for Git to assume there have been no changes made to `01-IntroducingGitInPractice.asciidoc`.

Solution

- 1 Change to the directory containing your repository; for example,
`cd /Users/mike/GitInPracticeRedux/`.

- 2 Run `git update-index --assume-unchanged 01-IntroducingGitInPractice.asciidoc`. There will be no output.

Git will ignore any changes made to `01-IntroducingGitInPractice.asciidoc`.

Discussion

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. This isn't generally a problem on normal filesystems on which Git can quickly query whether a file has been modified by checking the File Modified timestamp (rather than having to read the entire file and compare it).

`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; for example, `git update-index --assume-unchanged 00-Preface.asciidoc 01-IntroducingGitInPractice.asciidoc`.

The `git update-index` command has other complex options, but we'll only cover those around the "assume" logic. The rest of the behavior is better accessed through the `git add` command; it's a higher-level and more user-friendly way of modifying the state of the index.

Technique 27 Listing assumed-unchanged files

When you've told Git to assume no changes were made to particular files, it can be hard to remember which files were updated. In this case, you may end up modifying a file and wondering why Git doesn't seem to want to show you the changes. Additionally, you could forget that you made the changes and be confused as to why the state in your text editor doesn't seem to match the state that Git is seeing.

Problem

You wish for Git to list all the files that it has been told to assume haven't changed.

Solution

- 1 Change to the directory containing your repository; for example, `cd /Users/mike/GitInPracticeRedux/`.
- 2 Run `git ls-files -v`. The output should resemble the following.

Listing 3.11 Output: listing assumed-unchanged files

```
# git ls-files -v
```

```
H .gitignore
```

```
h 01-IntroducingGitInPractice.asciidoc
```

① Committed file



② Assumed-unchanged file



- ❶ shows that committed files are indicated by an uppercase H at the beginning of the line.
- ❷ shows that an assumed-unchanged file is indicated by a lowercase h tag.

Discussion

Like `git update-index`, `git ls-files -v` is a low-level command that you'll typically not run often. `git ls-files` without any arguments lists the files in the current directory that Git knows about, but the `-v` argument means it's followed by tags that indicate file state.

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.

Listing 3.12 Output: assumed-unchanged files

```
# git ls-files -v | grep '^[hsmrck?]' | cut -c 3-
01-IntroducingGitInPractice.asciidoc      ← Assumed-unchanged file
```

HOW DO PIPES, GREP, AND CUT WORK? Don't worry if you don't understand quite how Unix pipes, `grep`, and `cut` work; this book is about Git rather than shell scripting, after all! Feel free to use the command as is, as a quick way of listing files that are assumed to be unchanged. To learn more about these, I recommend the Wikipedia page on Unix filters: [http://en.wikipedia.org/wiki/Filter_\(Unix\)](http://en.wikipedia.org/wiki/Filter_(Unix)).

Technique 28 Stopping assuming files are unchanged

Usually, telling Git to assume there have been no changes made to a particular file is a temporary option; if you have to keep files changed in the long term, they should probably be committed. At some point, you'll want to tell Git to once again monitor any changes made to these files.

With the example I gave previously in technique 26, eventually the Rails configuration file change I had been testing was deemed to be successful enough that I wanted to commit it so my coworkers and the servers could use it. If I merely used `git add` to make a new commit, then the change wouldn't show up, so I had to make Git stop ignoring this particular change before I could make a new commit.

Problem

You wish for Git to stop assuming there have been no changes made to `01-IntroducingGitInPractice.asciidoc`.

Solution

- 1 Change to the directory containing your repository; for example,
`cd /Users/mike/GitInPracticeRedux/`.
- 2 Run `git update-index --no-assume-unchanged 01-IntroducingGitInPractice.asciidoc`. There will be no output.

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-IntroducingGitInPractice.asciidoc`. The output should resemble the following.

Listing 3.13 --no-assume-unchanged output

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

Git will notice any current or future changes made to `01-IntroducingGitInPractice.asciidoc`.

Discussion

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.

3.1 Summary

In this chapter, you learned the following:

- How to use `git mv` to move or rename files
- How to use `git rm` to remove files or directories
- How to use `git clean` to remove untracked or ignored files or directories
- How and why to create a `.gitignore` file
- How to (carefully) use `git reset --hard` to reset the working directory to the previously committed state
- How to use `git stash` to temporarily store and retrieve changes
- How to use `git update-index` to tell Git to assume files are unchanged

5

Advanced branching

This chapter covers

- Configuring `git merge`'s behavior
- Resolving merge conflicts
- Avoiding having to solve the same merge conflicts multiple times
- Creating a tag
- Generating a version number from previous tags
- Bringing individual commits from one branch to another
- Reverting a previous commit
- Listing what branches contain a given commit

When working as part of a team on a software project, you'll typically use branches to separate work between individuals, features, bug fixes, and software releases. You should already be able to perform some basic branching actions, such as creating, deleting, and merging a branch. This chapter will expand on those so you can improve your branching workflow to be more effective. Let's start by learning how to use some of the parameters provided by `git merge`.

Technique 33 Merging branches and always creating a merge commit

You learned in technique 14 how to perform a basic merge of two branches by using `git merge branchname`, where `branchname` is the name of the branch you wish to merge into the current branch.

Recall that a *merge commit* is one that has multiple parents and is displayed in GitX by the convergence of two or more branch *tracks*. `git merge` provides various options for merging branches without creating merge commits, using various strategies or resolving conflicts with a graphical merge tool.

WHY WOULD YOU WANT TO FORCE THE CREATION OF A MERGE COMMIT? Although fast-forward merges can sometimes be useful in some Git workflows, you should explicitly signify the merging of a branch even if it isn't necessary to do so. 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.

Let's start by setting up how to perform a merge that could be made without creating a merge commit: a fast-forward merge. Recall that a *fast-forward merge* means the incoming branch has the current branch as an ancestor. This means commits have been made on the incoming branch, but none have been made on the current branch since the incoming branch was branched from it.

You're creating a branch that can have a fast-forward merge. This is so when you create a merge commit, you know it was because it was specifically requested, not because it was required.

Let's create a branch that can be merged without a merge commit:

- 1 Change to the directory containing your repository: for example,
`cd /Users/mike/GitInPracticeRedux/`.
- 2 Run `git checkout -b chapter-spacing`.
- 3 Edit `01-IntroducingGitInPractice.asciidoc` to add a line break between chapters.
- 4 Run `git commit --message 'Add spacing between chapters' 01-Introducing-GitInPractice.asciidoc`. The output should resemble the following.

Listing 5.1 Output: fast-forward branch commit

```
# git commit --message 'Add spacing between chapters'
01-IntroducingGitInPractice.asciidoc

[chapter-spacing 4426877] Add spacing between chapters
1 file changed, 1 insertion(+)
```

You can see from figure 5.1 that you've created a new branch named `chapter-spacing` that can be merged without a merge commit into the master branch.

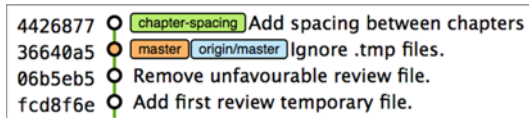


Figure 5.1 Local repository before merge without a merge commit

Problem

You wish to merge the `chapter-spacing` branch into the `master` branch and create a merge commit—not perform a fast-forward merge.

Solution

- 1 Change to the directory containing your repository: for example,
`cd /Users/mike/GitInPracticeRedux/.`
- 2 Run `git checkout master`.
- 3 Run `git merge --no-ff chapter-spacing`. You'll be prompted for a commit message, but you can accept the default. The output should resemble the following.

Listing 5.2 Output: forced merge commit

```
# git merge --no-ff chapter-spacing
```

```
Merge made by the 'recursive' strategy.
```

```
01-IntroducingGitInPractice.asciidoc | 1 +
```

```
1 file changed, 1 insertion(+)
```

← ① Merge type

← ② Diff summary

① shows that this was a merge (rather than a fast-forward) and therefore produced a merge commit. It used the *recursive* Git merge strategy (we'll discuss strategies more in the discussion section).

② shows a short summary of the changes made in this merge commit—all the differences between the `master` branch and the `chapter-spacing` branch.

You can now delete the merged `chapter-spacing` branch by running `git branch --delete chapter-spacing` from the `master` branch.

You've now merged the `chapter-spacing` branch into the `master` branch and forced a merge commit to be created.

Discussion

A merge commit has two parents: the previous commit on the current branch (`master` in this case) and the previous commit on the incoming branch (`chapter-spacing` in this case). You can see from figure 5.2 that GitX shows a merge commit differently from a fast-forward. Even when the `chapter-spacing` branch is deleted, the existence of a branch remains implied by the visual branching and merging in GitX.

In this case, where the branch contained a single commit, this may not be terribly useful. But on larger features, this explicit indication of branches can aid history visualization.

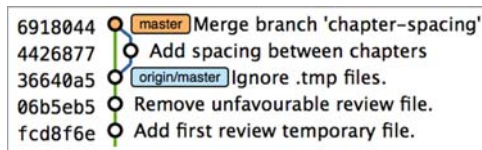


Figure 5.2 Local repository after `git merge --no-ff chapter-spacing`

`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 non-whitespace 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*

As mentioned previously, sometimes when you merge one branch into another, there will have been changes to the same part of the same file in both branches, and Git can't detect automatically which of these changes is the desired one to include. In this situation you have what's known as a *merge conflict*, which you'll need to resolve manually.

These situations tend to occur more often in software projects where multiple users are working on the same project at the same time. One user might make a bug fix to a file while another refactors it, and when the branches are merged, a merge conflict results.

Let's create a new branch and change the same files in both branches to produce a merge conflict:

- 1 Change to the directory containing your repository: for example,
`cd /Users/mike/GitInPracticeRedux/`.
- 2 Run `git checkout -b separate-files`.
- 3 Run `git mv 01-IntroducingGitInPractice.asciidoc 00-Preface.asciidoc`.
- 4 Cut the "Chapter 2" section from `00-Preface.asciidoc`, and paste it into a new file named `02-AdvancedGitInPractice.asciidoc`.
- 5 Cut the "Chapter 1" section from `00-Preface.asciidoc`, and paste it into a new file named `01-IntroducingGitInPractice.asciidoc`.

- 6 Run `git add`.
- 7 Run `git commit --message 'Separate files.'`. The output should resemble the following.

Listing 5.3 Output: committing separate files

```
# git commit --message 'Separate files.'

[separate-files 4320fad] Separate files.
3 files changed, 3 insertions(+), 4 deletions(-)
create mode 100644 00-Preface.asciidoc
create mode 100644 02-AdvancedGitInPractice.asciidoc
```

Now let's change the same file in the master branch:

- 1 Change to the directory containing your repository: for example,
`cd /Users/mike/GitInPracticeRedux/`.
- 2 Run `git checkout master`.
- 3 Edit `01-IntroducingGitInPractice.asciidoc` to add content for chapter 1.
- 4 Run `git commit --message 'Add Chapter 1 content.' 01-IntroducingGit-InPractice.asciidoc`. The output should resemble the following.

Listing 5.4 Output: committing chapter 1 content

```
# git commit --message 'Add Chapter 1 content.'
01-IntroducingGitInPractice.asciidoc

[master 7a04d8f] Add Chapter 1 content.
1 file changed, 3 insertions(+), 1 deletion(-)
```

After these edits, 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.

Listing 5.5 Current state on branches

```
# git show master:01-IntroducingGitInPractice.asciidoc

= Git In Practice
## Chapter 1
It is a truth universally acknowledged, that a single person in
possession of good source code, must be in want of a version control
system.

## Chapter 2
// TODO: write two chapters

# git show separate-files:01-IntroducingGitInPractice.asciidoc

## Chapter 1
// TODO: think of funny first line that editor will approve.
```

Figure 5.3 shows the current state of the master and separate-files branches in GitX.

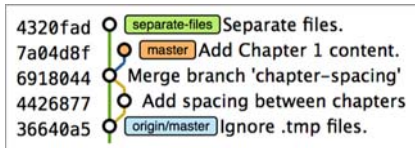


Figure 5.3 Local repository before merge-conflict resolution

Problem

You wish to merge the `separate-files` branch into the `master` branch and resolve the resulting merge conflict.

Solution

- 1 Change to the directory containing your repository: for example,
`cd /Users/mike/GitInPracticeRedux/`.
- 2 Run `git checkout master`.
- 3 Run `git merge separate-files`. The output should resemble the following.

Listing 5.6 Output: merge with a conflict

```
# git merge separate-files

Auto-merging 01-IntroducingGitInPractice.asciidoc
CONFLICT (content): Merge conflict in
  01-IntroducingGitInPractice.asciidoc
Automatic merge failed; fix conflicts and then commit the result.
```

- 1 shows Git attempting to find a way to solve the merge automatically using the default, recursive merge strategy.
- 2 shows that the merge strategy was unable to automatically solve the merge conflict, so it requires human intervention.

Now you need to edit `01-IntroducingGitInPractice.asciidoc` and solve the merge conflict. When you open the file, you'll see something resembling the following.

Listing 5.7 Before merge-conflict resolution

```
Unchanged line 1 ## Chapter 1
<<<<<<< HEAD 2 Incoming marker
It is a truth universally acknowledged, that a single person in 3 Incoming line
possession of good source code, must be in want of a version control
system.

## Chapter 2
// TODO: write two chapters 4 Branch separator
=====
// TODO: think of funny first line that editor will approve. 5 Current version
>>>>>> separate-files 6 Current marker
```

Recall this output and annotations from section 2.2:

- ❶ is provided for context.
- ❷ starts the current branch section containing the lines from the current branch (referenced by HEAD here).
- ❸ shows a line from the incoming branch.
- ❹ starts the section containing the lines from the incoming branch.
- ❺ shows a line from the current branch.
- ❻ ends the section containing the lines from the incoming branch (referenced by separate-files, the name of the branch being merged in).

You now need to edit the file so it has the correct version. In this case, this involves removing the chapter 2 section, because it was moved to another file in the separate-files branch, and using the new chapter 1 content that was entered in the master branch (here indicated by the HEAD section).

After editing, the file should resemble the following.

Listing 5.8 After merge-conflict resolution

```
## Chapter 1
It is a truth universally acknowledged, that a single person in
possession of good source code, must be in want of a version control
system.
```

Now that the merge conflict has been resolved, it can be marked as resolved with `git add` and then the merge commit committed. You don't need to run `git merge` again; you're still in the middle of a merge operation, which concludes when you `git commit`:

- ❶ Change to the directory containing your repository: for example,
`cd /Users/mike/GitInPracticeRedux/`.
- ❷ Run `git add 01-IntroducingGitInPractice.asciidoc`.
- ❸ Run `git commit`. Accept the default commit message. The output should resemble the following.

Listing 5.9 Output: committing the merge conflict

```
[master 725c33a] Merge branch 'separate-files'
```

You can run `git branch --delete separate-files` to delete the branch now that it's merged.

You've merged two branches and resolved a merge conflict.

Discussion

Merge commits have default commit message formats and slightly different diff output. Let's take a look at the merge commit by running `git show master`.

Listing 5.10 Output: merge commit

```
# git show master

commit 725c33ace6cd7b281c2d3b342ca05562d3dc7335
Merge: 7a04d8f 4320fad
Author: Mike McQuaid <mike@mikemcquaid.com>
Date: Sat Feb 1 14:55:38 2014 +0100

    Merge branch 'separate-files'

    Conflicts:
        01-IntroducingGitInPractice.asciidoc

diff --cc 01-IntroducingGitInPractice.asciidoc
index 6a10e85,848ed39..c9cda9c
--- a/01-IntroducingGitInPractice.asciidoc
+++ b/01-IntroducingGitInPractice.asciidoc
@@@ -1,8 -1,2 +1,4 @@@
- = Git In Practice 1
- == Chapter 1
- // TODO: think of funny first line that editor will approve.
+It is a truth universally acknowledged, that a single person in
+possession of good source code, must be in want of a version control
+system.
-
- == Chapter 2
- // TODO: write two chapters
```

← ① Merge subject

← ② Conflicted file

← ③ Incoming delete

④ Current delete

Current insert ⑤

- ① shows the default commit message subject for merge commits. It specifies the incoming branch name. It can be changed; but I prefer to leave it as is and add any additional information in the commit message body instead, so it's easily recognizable from the subject alone as a merge commit.
- ② shows a file that had conflicts to be resolved manually. Sometimes these conflicts may be resolved incorrectly, so this list is useful in spotting which files required resolution so they can be reviewed by other people later.
- ③ shows a line that was deleted in the incoming (*separate-files*) branch's commit(s). The `-` is in the first column as a result.
- ④ shows a line that was deleted in the current (*master*) branch's commit(s). The `-` is in the second column as a result.
- ⑤ shows a line that was inserted in the current (*master*) branch's commit(s). The `+` is in the second column as a result.

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 (*separate-files*), 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.

You can see from figure 5.4 that the changes from both branches are visible in the GitX output and that they're not always shown in chronological order. The Add Chapter 1 content commit occurs before the Separate files. commit even although it was made 3 minutes later.

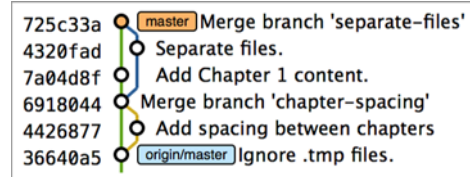


Figure 5.4 Local repository after merge-conflict resolution

5.2 Using a graphical merge tool

Instead of manually editing the contents of a file, you can instead run `git mergetool`, which runs a graphical merge tool such as `emerge`, `gvimdiff`, `kdifff3`, `meld`, `vimdiff`, `opendiff`, or `tortoisemerge`. Details of how to configure `git mergetool` to use your tool of choice are available by running `git help mergetool`.

Sometimes it can be helpful to use a graphical merge tool to be able to visualize conflicts graphically and understand how they relate to the changes that have been made by viewing them, say, side by side. Although I personally tend not to use these tools anymore, I found them useful when learning how to use version control.

You can also customize the tool that is used to specify your own merge tools. Figure 5.5 shows the `opendiff` tool provided with OS X being used to resolve the previous merge conflict.

WHO SHOULD RESOLVE MERGE CONFLICTS? In Git, the person who makes a merge (runs the `git merge` command) should always be the person who resolves a merge conflict. This may differ from other version control systems. Additionally, this may sometimes mean that if a conflict resolution requires a particular member of a team to be able to pick the correct resolution, the `git merge` should be done by this person.

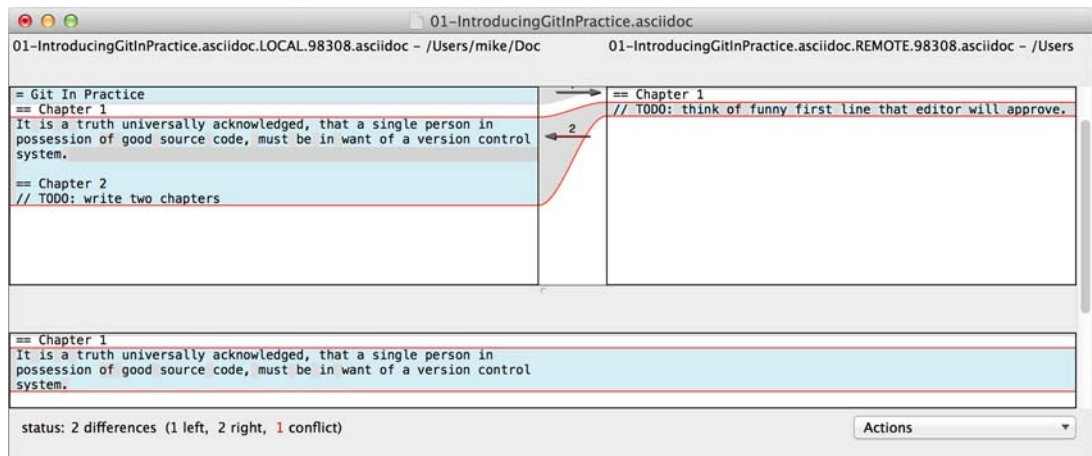


Figure 5.5 `opendiff` merge-conflict resolution

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

You may find yourself in a situation where you have a long-running branch that you have to keep merging in another branch, and you get the same merge conflicts every time. It can be frustrating to have to manually resolve the same merge conflict multiple times; after all, isn't repeatedly performing boring tasks what computers are good for?

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. In short, you only need to solve a particular merge conflict once. I always tend to enable this when I use Git, because it runs automatically to ensure that I don't need to solve the same merge conflicts multiple times if I'm doing something like repeatedly merging the same branch, which produces the same conflict.

When `git rerere` is enabled, it stores the changes before a merge conflict and compares them to after the merge conflict was resolved. This is used to fingerprint a particular merge conflict based on the entire contents of the conflict (the changes from both branches). This fingerprint is then used whenever there's another merge conflict, to compare against all the previously resolved merge conflicts. If a merge conflict is detected to be the same, then `git rerere` reuses the previously recorded merge-conflict resolution and edits the files as if you had resolved it manually. You still need to use `git add` to mark the merge conflict as resolved, however; this is in case you've decided to resolve this merge conflict in a slightly different way, so Git gives you a chance to edit it.

Let's learn how to set up `git rerere`.

Problem

You want to set up `git rerere` to integrate with the merge workflow so you don't need to repeatedly resolve the same merges.

Solution

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.

Discussion

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.

Listing 5.11 `rerere` merge-conflict storage

```
# git commit
Recorded resolution for '01-IntroducingGitInPractice.asciidoc'.
[master 725c33a] Merge branch 'separate-files'
```

rerere
storage

git rerere has been run by git commit to store the conflict and resolution so it can recall the same resolution when it sees the same conflict.

The output is as follows if the same conflict is seen again.

Listing 5.12 rerere merge-conflict retrieval

```
# git merge separate-files

Auto-merging 01-IntroducingGitInPractice.asciidoc
CONFLICT (content): Merge conflict in
  01-IntroducingGitInPractice.asciidoc
Resolved '01-IntroducingGitInPractice.asciidoc' using
  previous resolution.
Automatic merge failed; fix conflicts and then commit the result.
```

rerere
retrieval

git rerere has again been run by git merge to retrieve the resolution for the identical conflict. You still need to run git add to accept the conflict, and you can use git diff or edit the file to ensure that the resolution was as expected and desired.

HOW CAN YOU MAKE GIT RERERE FORGET AN INCORRECT RESOLUTION? 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

Remember refs from section 1.7? A 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.

For example, if you were releasing version 1.3 of your software project, you'd tag the commit that you release to customers as v1.3 to store that version for later use. Then if a customer later complained about something being broken in v1.3, you could check out that tagged commit and test against it, confident that you were using the same version of the software that the customer was. This is one of the reasons you shouldn't modify tags; once you've released a version to customers, if you want to update it, you'll likely release a new version such as 1.4 rather than changing the definition of 1.3.

Figure 5.6 shows the current state of the master branch in GitX before the tag has been created.

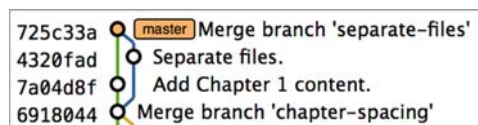


Figure 5.6 Local repository before git tag

Problem

You wish to tag the current state of the `GitInPracticeReduxmaster` branch as version `v0.1`.

Solution

- 1 Change to the directory containing your repository: for example,
`cd /Users/mike/GitInPracticeRedux/`.
- 2 Run `git checkout master`.
- 3 Run `git tag v0.1`. There will be no output.
- 4 Run `git tag`. The output should resemble the following.

Listing 5.13 Output: tag listing

```
# git tag
```

```
v0.1 ← 1 Version tag
```

1 shows that there is a tag named `v0.1` in the local repository. All tags in the current repository (not just the current branch) are listed by `git tag`.

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

Discussion

You can see from figure 5.7 that after `git tag`, there's a new `v0.1` ref on the latest commit on the `master` branch (in the `GitX` interface, this is yellow). This indicates that this commit has been tagged `v0.1`.

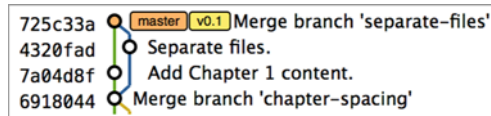


Figure 5.7 Local repository after `git tag`

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 useful for 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 commit.

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*

You've seen that `git tag` can be used to identify certain commits as released versions of a project. I'm a passionate advocate of continuous integration systems, and I've worked on desktop software projects with semantic versioning (such as 1.0.0). On these projects, I've set up continuous integration systems to create installers of the software on every commit to the master branch.

But some software has an About screen that displays the software's version. In this case, I'd like to have a version number generated that makes sense but doesn't rely on auto-generating a tag for each version of the software and is sortable with some information about the current version of the software. Something like `v0.1-1-g0a5e328` would be preferential to a short revision like `g0a5e328`.

The expected version number would be `v0.1`, given that has just been tagged, so let's make another modification to the `GitInPracticeRedux` repository and generate a version number for the new, untagged commit:

- 1 Change to the directory containing your repository: for example
`cd /Users/mike/GitInPracticeRedux/`.
- 2 Add some content to the `00-Preface.asciidoc` file.
- 3 Run `git commit --message 'Add preface text.' 00-Preface.asciidoc`. The output should resemble the following.

Listing 5.14 *Output: committing the preface*

```
# git commit --message 'Add preface text.'
[master 0a5e328] Add preface text.
1 file changed, 1 insertion(+)
```

Problem

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

Solution

- 1 Change to the directory containing your repository: for example,
`cd /Users/mike/GitInPracticeRedux/`.
- 2 Run `git describe --tags`. The output should resemble the following.

Listing 5.15 Output: tag describe

```
# git describe --tags  
v0.1.1-g0a5e328 ← ① Generated version
```

① 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'd run `git describe --tags` when on the previous commit (the `v0.1` tag), it would've output `v0.1`.

You've generated a version number based on the existing tags in the repository.

Discussion

If `git describe` is passed a ref, it generates the version number for that particular commit. For example, `git describe --tags v0.1` outputs `v0.1`, and `git describe --tags 0a5e328` outputs `v0.1.1-g0a5e328`.

If you wish to generate the long-form versions for tagged commits, you can pass the `--long` flag. For example, `git describe --tags --long v0.1` outputs `v0.1-0-g725c33a`.

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***

Sometimes you may wish to include only a single commit from a branch onto the current branch rather than merging the entire branch. For example, you may want to back-port a single bug-fix commit from a development branch into a stable release

branch. You could do this by manually creating the same change on that branch, but a better way is to use the tool that Git provides: `git cherry-pick`.

Let's create a new branch based off the `v0.1` tag called `v0.1-release` so you have something to `cherry-pick`:

- 1 Change to the directory containing your repository: for example,
`cd /Users/mike/GitInPracticeRedux/`.
- 2 Run `git checkout -b v0.1-release v0.1`.
- 3 Add some content to the `02-AdvancedGitInPractice.asciidoc` file.
- 4 Run `git commit --message 'Advanced practice technique.' 02-Advanced-GitInPractice.asciidoc`. The output should resemble the following.

Listing 5.16 Output: committing the release branch

```
# git commit --message 'Advanced practice technique.'
02-AdvancedGitInPractice.asciidoc

[v0.1-release dfe2377] Advanced practice technique.
1 file changed, 1 insertion(+), 1 deletion(-)
```

Problem

You wish to `cherry-pick` a commit from the `v0.1-release` branch to the master branch.

Solution

- 1 Change to the directory containing your repository: for example,
`cd /Users/mike/GitInPracticeRedux/`.
- 2 Run `git checkout master`.
- 3 Run `git cherry-pick v0.1-release`. The output should resemble the following.

Listing 5.17 Output: commit cherry-pick

```
# git cherry-pick v0.1-release

[master c18c9ef] Advanced practice technique. ← ❶ Commit summary
1 file changed, 1 insertion(+), 1 deletion(-)
```

- ❶ shows the result of the `cherry-pick` operation. Note that this is the same as the output for the previous `git commit` command, with one difference: the SHA-1 has changed.

WHY DOES THE SHA-1 CHANGE ON A CHERRY-PICK? Recall that the SHA-1 of a commit is based on its tree and metadata (which includes the parent commit SHA-1). Because the resulting master branch `cherry-picked` commit has a different parent than the commit that was `cherry-picked` from the `v0.1-release` branch, the commit SHA-1 differs also.

You've cherry-picked a commit from the v0.1-release branch to the master branch.

Discussion

`git cherry-pick` (like many other Git commands) can take a ref as the parameter rather than only a specific commit. As a result, you could have interchangeably used `git cherry-pick dfe2377` (where `dfe2377` is the most recent commit on the v0.1-release branch) in the previous example with the same result. You can pass multiple refs to `cherry-pick`, and they will be cherry-picked onto the current branch in the order requested.

HOW MANY COMMITS SHOULD YOU CHERRY PICK? Cherry-picking is best used for individual commits that may be out of sequence. The classic use case highlighted earlier is back-porting bug fixes from a development branch to a stable branch. When this is done, it's effectively duplicating the commits (rather than sharing them as with a merge). If you find yourself wanting to cherry-pick the entire contents of a branch, you'd be better off merging it instead.

`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 cherry-picked 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`.

WHEN WOULD YOU SIGN OFF A COMMIT? Signing off a commit is generally used in projects to indicate that a commit was checked by someone else before being included. I'm a maintainer of the Homebrew open source project and use signing off to indicate to other maintainers that I was the one who merged this commit. This information is also included as the Author meta-data in the commit, but the sign-off makes it more readily accessible. The same process could be used in companies when a developer reviews the work of another and wants to signify this in a commit message.

Technique 39 *Reverting a previous commit: git revert*

You may occasionally make a commit that you regret. You'll then want to undo the commit until you can fix it so it works as intended.

In Git you can rewrite history to hide such mistakes (as you'll learn later in technique 42), but this is generally considered bad practice if you've already pushed a commit publicly. In these cases, you're better off instead using `git revert`.

Problem

You wish to revert a commit to reverse its changes.

Solution

- 1 Change to the directory containing your repository: for example,
`cd /Users/mike/GitInPracticeRedux/`.
- 2 Run `git checkout master`.
- 3 Run `git revert c18c9ef`. You're prompted for a message. Accept the default. The output should resemble the following.

Listing 5.18 Output: revert

```
# git revert c18c9ef

[master 3e3c417] Revert "Advanced practice technique." ← Revert subject
1 file changed, 1 insertion(+), 1 deletion(-)
```

To view the revert in more depth, run `git show 3e3c417`.

Listing 5.19 Output: revert show

```
# git show 3e3c417

commit 3e3c417e90b5eb3c04962618b238668d1a5dc5ab
Author: Mike McQuaid <mike@mikemcquaid.com>
Date: Sat Feb 1 20:26:06 2014 +0000

    Revert "Advanced practice technique." ← ① Revert subject

    This reverts commit c18c9ef9adc73cc1da7238ad97ffb50758482e91. ← ② Reversed diff

diff --git a/02-AdvancedGitInPractice.asciidoc
    b/02-AdvancedGitInPractice.asciidoc
index 0e0765f..7eb5017 100644
```

```

--- a/02-AdvancedGitInPractice.asciidoc
+++ b/02-AdvancedGitInPractice.asciidoc
@@ -1,2 +1,2 @@
  == Chapter 2
-Practice doesn't make perfect; perfect practice makes perfect!
+// TODO: write two chapters

```

3 Revert
message

- 1** shows the reverted commit's subject prefixed with `Revert`. This should describe what has been reverted fairly clearly; it can be edited on commit if it doesn't.
- 2** shows the body of the reverted commit, which includes the full SHA-1 of the commit that was reverted.
- 3** shows the diff of the new commit. It's the exact opposite of the diff of the commit that was reverted.

You've reverted a commit to reverse its changes.

Discussion

`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`

If you have a workflow in which you don't merge your commits to other branches but rather have another person do it, you may wish to see which of your commits has been merged to another branch. Git has a tool to do this: `git cherry`.

Let's make another commit on the `v0.1-release` branch first:

- 1** Change to the directory containing your repository: for example,
`cd /Users/mike/GitInPracticeRedux/`.
- 2** Run `git checkout v0.1-release`.
- 3** Add some content to the `00-Preface.asciidoc` file.
- 4** Run `git commit --message 'Add release preface.' 00-Preface.asciidoc`.
The output should resemble the following.

Listing 5.20 Output: committing the release preface

```

[v0.1-release a8200e1] Add release preface.
1 file changed, 1 insertion(+)

```

Problem


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

Solution

- 1 Change to the directory containing your repository: for example,
`cd /Users/mike/GitInPracticeRedux/.`
- 2 Run `git checkout v0.1-release`.
- 3 Run `git cherry --verbose master`. The output should resemble the following.

Listing 5.21 Output: cherry

```
# git cherry --verbose master
- dfe2377f00bb58b0f4ba5200b8f4299d0bfeeb5d Advanced practice technique.
+ a8200e1407d49e37baad47da04c0981f43d7c7ff Add release preface.
```



- ① is prefixed with `-` and shows a commit that has been already included into the master branch.
- ② is prefixed with `+` and shows a commit that hasn't yet been included into the master branch.

You've seen which commits remain unmerged from the master branch.

Discussion

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: for example,

```
- dfe2377f00bb58b0f4ba5200b8f4299d0bfeeb5d.
```

When you learn about rebasing later in technique 43, you'll see how `git cherry` can be useful for showing what commits will be kept or dropped after a rebase operation.

5.3 Summary

In this chapter you learned the following:

- How to use `git merge`'s options to perform different types of merges
- How to resolve merge conflicts
- How to use `git rerere` to repeatedly replay merge-conflict resolutions
- How to use `git tag` to tag commits
- How to use `git describe` to generate version numbers for commits
- How to use `git cherry-pick` to bring individual commits from one branch to another
- How to use `git revert` to reverse individual commits
- How to use `git cherry` to list what commits remain unmerged on a branch

13

Merging vs. rebasing

This chapter covers

- Using CMake's branching and merging strategy to manage contributions
- Using Homebrew's rebasing and squashing strategy to manage contributions
- Deciding what strategy to use for your project

As discussed in technique 14 and technique 43, merging and rebasing are two strategies for updating the contents of one branch based on the contents of another. Merging joins the history of two branches together with a merge commit (a commit with two parent commits); and rebasing creates new, reparented commits on top of the existing commits.

Why are there two strategies for accomplishing essentially the same task? Let's find out by comparing the Git history of two popular open source projects and their different branching strategies.

13.1 CMake's workflow

CMake is a cross-platform build-system created by Kitware. It has many contributors both inside and outside Kitware; most contributions are among those with direct push access to the Kitware Git repository.

CMake's Git repository is available to access at <http://cmake.org/cmake.git>. It's also mirrored on GitHub at <https://github.com/Kitware/CMake> if you'd rather browse or clone it from there. Please clone it and examine it while reading this chapter.

CMake makes heavy use of branching and merges. Several of the branches visible or implied in figure 13.1 are as follows:

- **next**—Shown in the figure as `origin/next`. This is an *integration branch* used for integration of *feature branches* (also known as *topic branches*) when developing a new version of CMake. `master` is merged in here regularly to fix merge conflicts.
- **nightly**—Shown in the figure as `origin/nightly`. It follows the `next` branch and is updated to the latest commit on `next` automatically at 01:00 UTC every day. `nightly` is used by automated nightly tests to get a consistent version for each day.
- **master**—Seen in figure indirectly; merged in the Merge 'branch' master into `next` commit. This is an *integration branch* that is always kept ready for a new release; release branches are merged into here and then deleted. New feature branches are branched off of `master`.
- **Feature branches**—Seen in the figure as Merge topic '...' into `next` commits. These are used for development of all bug fixes and new features. All new commits (except merge commits) are made on feature branches. They're merged into `next` for integration testing and `master` for permanent inclusion and can then be deleted.

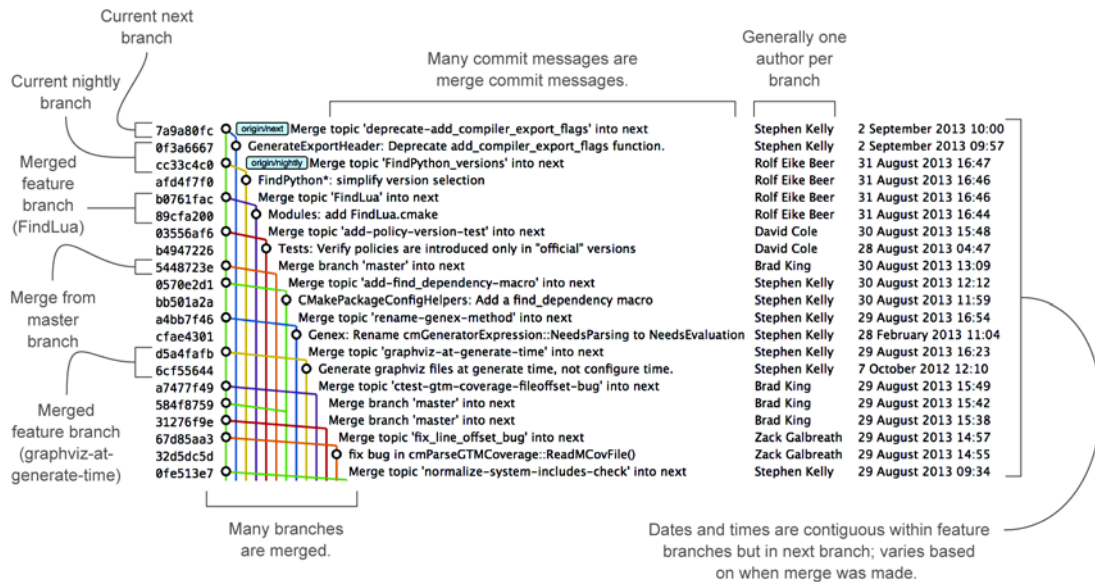


Figure 13.1 CMake repository history

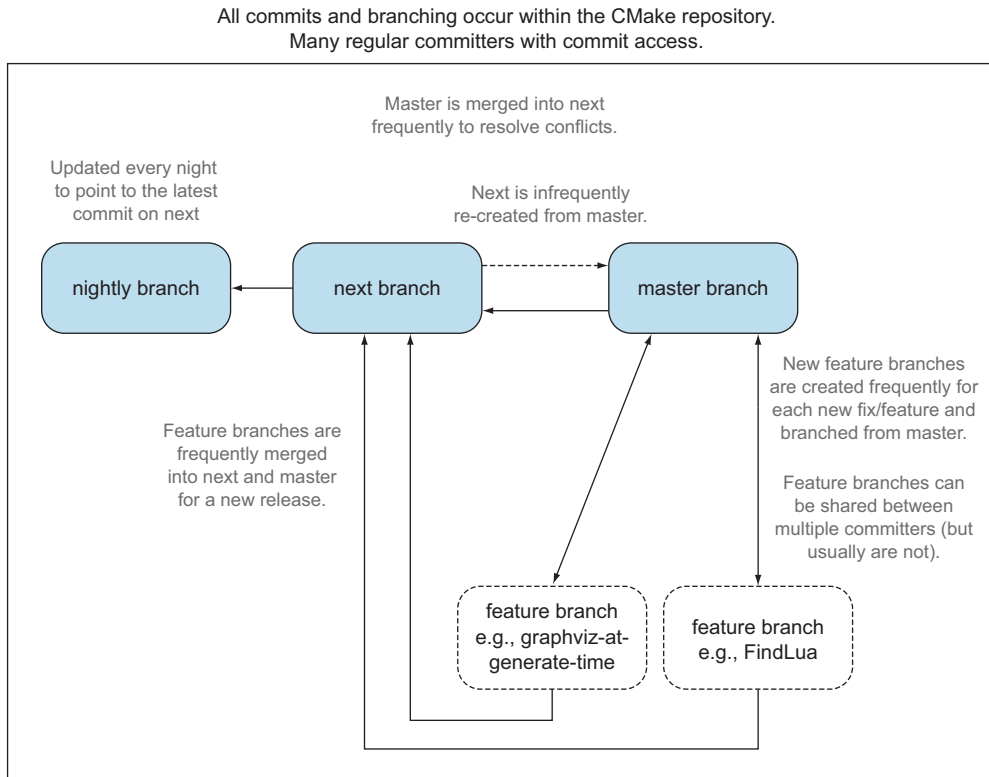


Figure 13.2 CMake branch/merge workflow

The merging of master into next is done immediately after merging any feature branch to master. This ensures that any merge conflicts between master and next are resolved quickly in the next branch. The regular merging of feature branches into next allows integration testing before a new release is prepared and provides context for individual commits; the branch name used in the merge commit helps indicate what feature or bug the commit was in relation to.

Figure 13.2 focuses on the interactions between branches in the CMake workflow (rather than the interactions between commits and branches in figure 13.1). For a new commit to end up in master, a new feature branch needs to be created, commits must be made on it, the feature branch must be merged to the next branch for integration testing, and finally the feature branch must be merged to master and deleted.

13.1.1 Workflow commands

The following commands are used by CMake developers to clone the repository, create new branches for review, and merge them to next to be tested, and by CMake core maintainers to finally merge them into master.

These steps set up the CMake repository on a local machine:

- 1 Clone the fetch-only CMake Git repository with `git clone http://cmake.org/cmake.git`.
- 2 Add the pushable staging repository with `git remote add stage git@cmake.org:stage/cmake.git`. The staging repository is used for testing and reviewing branches before they're ready to be merged. CMake developers are given push access to it, but only CMake core maintainers have push access to the main repository.

These commands make a new branch and submit it for review:

- 1 Fetch the remote branches with `git fetch origin`.
- 2 Branch from `origin/master` with `git checkout -b branchname origin/master`.
- 3 Make changes and commit them with `git add` and `git commit`.
- 4 Push the branch to the staging repository with `git push --set-upstream stage branchname`.
- 5 Post an email to the CMake mailing list (www.cmake.org/mailman/listinfo/cmake-developers) to ask other CMake developers for review and feedback of the changes.

These steps merge a branch for nightly testing:

- 1 Fetch the remote branches with `git fetch stage`.
- 2 Check out the next branch with `git checkout next`.
- 3 Merge the remote branch with `git merge stage/branchname`.
- 4 Push the next branch with `git push`.

CMake developers perform these steps with the `stage` command over SSH by running `ssh git@cmake.org stage cmake merge -b next branchname`.

These steps make changes based on feedback from other CMake developers:

- 1 Check out the branch with `git checkout branchname`.
- 2 Make changes and commit them with `git add` and `git commit`.
- 3 Push the new commits to the staging repository with `git push`.
- 4 Post another email to the CMake mailing list (www.cmake.org/mailman/listinfo/cmake-developers).

These steps allow a CMake core maintainer to merge a branch into master after successful review:

- 1 Fetch the remote branches with `git fetch stage`.
- 2 Check out the master branch with `git checkout master`.
- 3 Merge the remote branch with `git merge stage/branchname`.
- 4 Push the master branch with `git push`.

CMake core maintainers perform these steps with the `stage` command over SSH by running `ssh git@cmake.org stage cmake merge -b master branchname`.

13.2 Homebrew's workflow

Homebrew is a package manager for OS X. It has thousands of contributors but a very small number of maintainers with commit access to the main repository (five at the time of writing).

Homebrew's main Git repository is available to access at <https://github.com/Homebrew/homebrew>. Please clone it and examine it while reading this chapter.

Homebrew has very few merge commits in the repository (remember that *fast-forward merges* don't produce merge commits). In figure 13.3, you can see that the history is entirely continuous despite multiple commits in a row from the same author and noncontinuous dates. Branches are still used by individual contributors (with and without push access to the repository), but branches are rebased and squashed before being merged. This hides merge commits, evidence of branches, and temporary commits (for example, those that fix previous commits on the same branch) from the master branch.

Figure 13.4 focuses on the branches and repositories in the Homebrew workflow. New commits can end up on master by being directly committed by those with main repository access, by a feature branch being squashed and picked from a forked repository or, very rarely, through a major refactor branch being merged.

On the infrequent occasions when a major refactor branch is needed on the core repository (say, for heavy testing of the major refactor), it's kept as a branch in the main repository and then merged. This branch isn't used by users but may be committed to and tested by multiple maintainers.

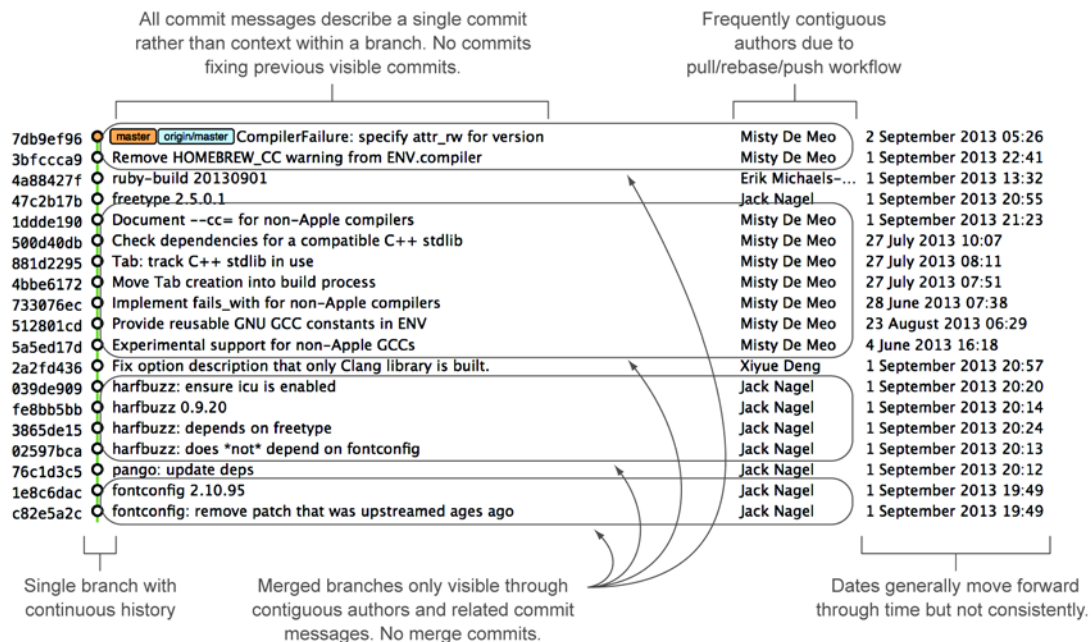


Figure 13.3 Homebrew repository history

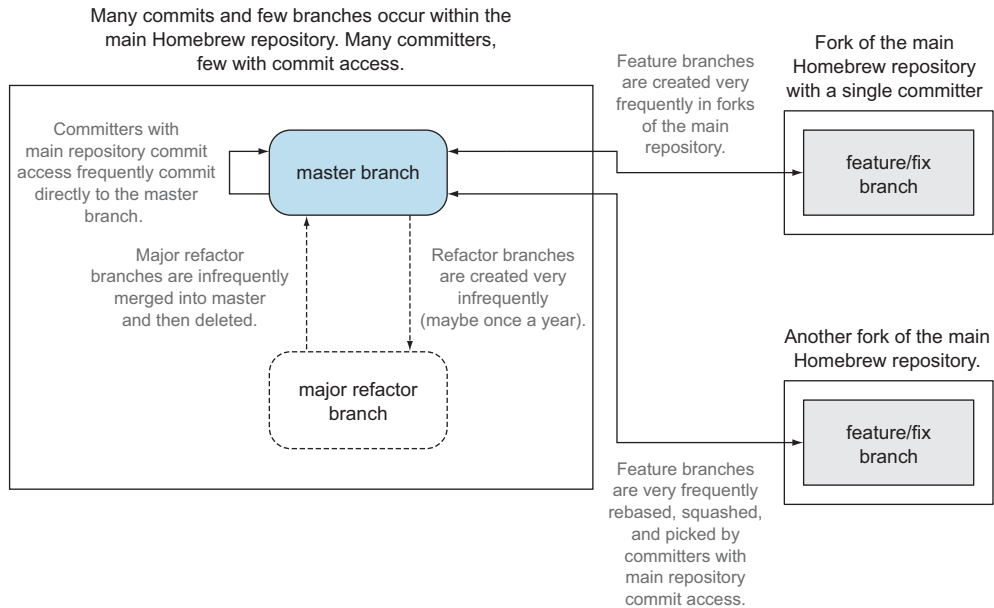


Figure 13.4 Homebrew's branch/rebase/squash workflow

13.2.1 Workflow commands

The following commands are used by Homebrew contributors to clone the repository, create new branches, and issue pull requests, and by Homebrew maintainers to finally merge them into master.

These commands set up the Homebrew repository on the local machine:

- 1 Clone the fetch-only Homebrew Git repository with `git clone https://github.com/Homebrew/homebrew.git`.
- 2 Fork the Homebrew repository on GitHub. This creates a pushable, personal remote repository. This is needed because only Homebrew maintainers have push access to the main repository.
- 3 Add the pushable forked repository with `git remote add username https://github.com/username/homebrew.git`.

These commands make a new branch and submit it for review:

- 1 Check out the master branch with `git checkout master`.
- 2 Retrieve new changes to the master branch with `git pull --rebase` (or Homebrew's `brew update` command, which calls `git pull`).
- 3 Branch from master with `git checkout -b branchname origin/master`.
- 4 Make changes and commit them with `git add` and `git commit`.
- 5 Push the branch to the fork with `git push --set-upstream username branchname`.
- 6 Create a *pull request* on GitHub, requesting review and merge of the branch.

These commands make changes based on feedback:

- 1 Check out the branch with `git checkout branchname`.
- 2 Make changes and commit them with `git add` and `git commit`.
- 3 Squash the new commits with `git rebase --interactive origin/master`.
- 4 Update the remote branch and the pull request with `git push --force`.

These commands allow a Homebrew maintainer to merge a branch into master:

- 1 Check out the master branch with `git checkout master`.
- 2 Add the forked repository and cherry-pick the commit with `git add remote username https://github.com/username/homebrew.git`, `git fetch username`, and `git merge username/branchname`. Alternatively, some maintainers (including me) use Homebrew's `brew pull` command, which pulls the contents of a pull request onto a local branch by using patch files rather than fetching from the forked repository.
- 3 Rebase, reword, and clean up the commits on master with `git rebase --interactive origin/master`. It's common for Homebrew maintainers to edit or squash commits and rewrite commit messages but preserve the author metadata so the author retains credit. Often a commit will be edited to contain a string like "Closes #123", which automatically closes the pull request numbered 123 when the commit is merged to master. This was covered in greater detail in chapter 10.
- 4 Push the master branch with `git push`.

13.3 **CMake workflow pros and cons**

CMake's approach makes it easy to keep track of what feature branches have been merged, when they were merged, and by whom. Individual features and bug fixes live in separate branches and are integrated only when and where it makes sense to do so. Individual commits and evidence of branches (but not the branches themselves) are always kept in history for future viewing. Feature branches are tested individually, and then integration testing is done in the next branch. When a feature branch is deemed to be in a sufficiently stable state, it's merged into the master branch and deleted. This ensures that the master branch is always stable and kept ready for a release.

When developing desktop software like CMake that ships binary releases, having a very stable branch is important; releases are a formal, time-consuming process, and updates can't be trivially pushed after release. Thus it's important to ensure that testing is done frequently and sufficiently before releasing.

CMake's approach produces a history that contains a lot of information but, as seen from the plethora of lines in figure 13.1, can be hard to follow. Merge commits are frequent, and commits with actual changes are harder to find as a result. This can make reverting individual commits tricky; using `git revert` on a merge commit is hard because Git doesn't know which side of the merge it should revert to. In addition, if you revert a merge commit, you can't easily re-merge it.

There are also potential trust issues with CMake’s approach. Everyone who wants to create a feature branch needs commit access to the CMake repository. Because Git and Git-hosting services don’t provide fine-grained access control (such as restricting access to particular branches), and because CMake’s Git workflow doesn’t rewrite history, anyone with commit access could, for example, make commits directly to the master branch and circumvent the process. Everyone who commits to CMake needs to be made aware of the process and trusted not to break or circumvent it. Kitware protects against process violations with rewriting and server-side checks. But this requires complex setup and server configuration and a willingness to rewrite pushed branches to fix mistakes.

13.4 Homebrew workflow pros and cons

A major benefit of Homebrew’s approach should be evident from figure 13.3: the history is simple. The master branch contains no direct merges, so ordering is easy to follow. Commits contain concise descriptions of exactly what they do, and there are no commits that are fixing previous ones. Every commit communicates important information.

As a result of commits being squashed, it’s also easy to revert individual commits and, if necessary, reapply them at a later point. Homebrew doesn’t have a release process (the top of the master branch is always assumed to be stable and delivered to users), so it’s important that changes and fixes can be pushed quickly rather than having a stabilization or testing process.

WHY IS A READABLE HISTORY IMPORTANT FOR HOMEBREW? Readable history is an important feature of Homebrew’s workflow. Homebrew uses Git not just as a version control system for developers, but also as an update delivery mechanism for users. Presenting users with a more readable history allows them to better grasp updates to Homebrew with basic Git commands and without understanding merges.

Homebrew’s workflow uses multiple remote repositories. Because only a few people have commit access to the core repository, their approach is more like that of Linus on the Git project (as discussed in section 1.1), often managing and including commits from others more than making their own commits. Many commits made to the repository are made by squashing and merging commits from forks into the master branch of the main repository. The squashing means any fixes that needed to be made to the commit during the pull request process won’t be seen in the master branch and each commit message can be tailored by the core team to communicate information in the best possible way.

This workflow means only those on the core team can do anything dangerous to the main repository. Anyone else’s commits must be reviewed before they’re applied. This puts more responsibility on the shoulders of the core team, but other contributors to Homebrew only need to know how to create a pull request and not how to do stuff like squash or merge commits.

Unfortunately, Homebrew's approach means most branch information is (intentionally) lost. It's possible to guess at branches from multiple commits with related titles and/or the same author for multiple commits in a row, but nothing explicit in the history indicates that a merge has occurred. Instead, metadata is inserted into commit messages stating that a commit was signed-off by a particular core contributor and which pull request (or issue) this commit related to.

13.5 *Picking your strategy*

Organizations and open source projects vary widely in their branching approaches. When picking between a branch-and-merge or a branch-rebase-and-squash strategy, it's worth considering the following:

- If all the committers to a project are trusted sufficiently and can be educated on the workflow, then giving everyone access to work on a single main repository may be more effective. If committers' Git abilities vary dramatically and some are untrusted, then using multiple Git repositories and having a review process for merges between them may be more appropriate.
- If your software can release continuous, quick updates (like a web application) or has a built-in updater (like Homebrew), then focusing development on a single (`master`) branch is sensible. If your software has a more time-consuming release process (such as desktop or mobile software that needs to be compiled and perhaps even submitted to an app store for review), then working across many branches may be more suitable. This applies even more if you have to actively support many released versions of the software simultaneously.
- If it's important to be able to trivially revert merged changes on a branch (and perhaps re-merge them later), then a squashing process may be more effective than a merging process.
- If it's important for the history to be easily readable in tools such as GitX and gitk, then a squashing process may be more effective. Alternatively, a merging process can still be done, but with less frequent merges so each merge contains at least two or more commits. This ensures that the history isn't overwhelmed with merge commits.

There are various other considerations you could take into account, but these are a good starting point. You may also consider creating your own blended approach that uses merging and squashing in different situations.

Regardless of which workflow you decide is best for your project, it's important to try to remain consistent: not necessarily across every branch (for example, it might be reasonable to always make merge commits in `master` but always rebase branches on top of other branches), but across the repository. This should ensure that, whatever strategy is adopted, the history will communicate something about the project's development process and new committers can look at the history for an example of what their workflow should be like.

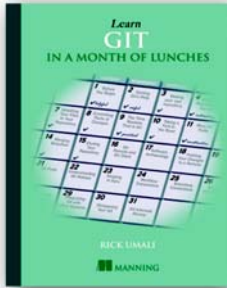
WHAT IS THE AUTHOR'S PREFERRED APPROACH? Although I've committed to both projects, most of my open source time is spent working on Homebrew. It will therefore probably come as no surprise to hear that I prefer Homebrew's approach. Maintaining a simple and readable history has frequently paid off in terms of quickly being able to `git bisect` or `git revert` problematic commits. Also, I prefer software-release processes that favor lots of small updates rather than fewer, large updates. I think these processes are easier to test, because they encourage incremental improvements rather than huge, sweeping changes.

13.6 Summary

In this chapter you learned the following:

- How CMake uses multiple branches to keep features developed in separation
- How Homebrew uses a single branch to release continuous updates to users
- How merging allows you to keep track of who added commits, when, and why
- How rebasing and squashing allow you to maintain a cleaner history and eliminate commits that may be irrelevant

RELATED MANNING TITLES



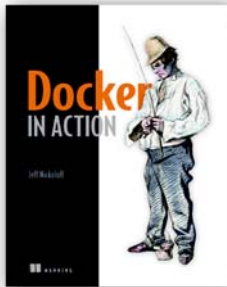
Learn Git in a Month of Lunches

by Rick Umali

ISBN: 9781617292415

375 pages, \$39.99

August 2015



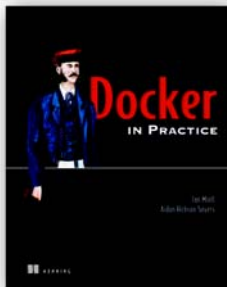
Docker in Action

by Jeff Nickoloff

ISBN: 9781633430235

300 pages, \$49.99

December 2015



Docker in Practice

by Ian Miell and Aidan Hobson Sayers

ISBN: 9781617292729

275 pages, \$44.99

November 2015



OpenStack in Action

by V. K. Cody Bumgardner

ISBN: 9781617292163

375 pages, \$54.99

October 2015

For ordering information go to www.manning.com