

Mike McQuaid
Foreword by Scott Chacon



Git

IN PRACTICE

INCLUDES 66 TECHNIQUES

Git in Practice

Git in Practice

MIKE McQUAID



MANNING
SHELTER ISLAND

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: orders@manning.com

©2015 by Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

© Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15 percent recycled and processed without the use of elemental chlorine.

 Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964

Development editor: Dan Maharry
Technical development editor: Glenn Burnside
Copyeditor: Benjamin Berg
Proofreader: Tiffany Taylor
Typesetter: Dottie Marsico
Cover designer: Marija Tudor

ISBN 9781617291975
Printed in the United States of America
1 2 3 4 5 6 7 8 9 10 – EBM – 19 18 17 16 15 14

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

contents

foreword *xiii*
preface *xv*
acknowledgments *xvii*
about this book *xviii*
about the cover illustration *xxii*

PART 1 INTRODUCTION TO GIT1

1 *Local Git* *3*

- 1.1 Why do programmers use Git? 3
- 1.2 Initial setup 4
 - TECHNIQUE 1 Creating a repository: git init 5
- 1.3 .git subdirectory 6
- 1.4 Creating a new commit: git add, git commit 7
 - TECHNIQUE 2 Building a new commit in the index staging area: git add 8
 - TECHNIQUE 3 Committing changes to files: git commit 10
 - TECHNIQUE 4 Viewing history: git log, gitk, gitx 13
- 1.5 Rewriting history 18
 - TECHNIQUE 5 Viewing the differences between commits: git diff 18
- 1.6 Diff formats 21

- 1.7 Refs 22
- 1.8 Summary 23

2 *Remote Git* 24

- TECHNIQUE 6 Adding a remote repository: `git remote add` 24
- 2.1 Authoritative version storage 27
 - TECHNIQUE 7 Pushing changes to a remote repository: `git push` 28
 - TECHNIQUE 8 Cloning a remote/GitHub repository onto your local machine: `git clone` 30
 - TECHNIQUE 9 Pulling changes from another repository: `git pull` 32
 - TECHNIQUE 10 Fetching changes from a remote without modifying local branches: `git fetch` 35
 - TECHNIQUE 11 Creating a new local branch from the current branch: `git branch` 37
 - TECHNIQUE 12 Checking out a local branch: `git checkout` 40
 - TECHNIQUE 13 Pushing a local branch remotely 42
 - TECHNIQUE 14 Merging an existing branch into the current branch: `git merge` 44
- 2.2 Merge conflicts 46
- 2.3 Rebasing 47
 - TECHNIQUE 15 Deleting a remote branch 47
 - TECHNIQUE 16 Deleting the current local branch after merging 49
- 2.4 Summary 50

PART 2 GIT ESSENTIALS51

3 *Filesystem interactions* 53

- TECHNIQUE 17 Renaming or moving a file: `git mv` 54
- TECHNIQUE 18 Removing a file: `git rm` 55
- TECHNIQUE 19 Resetting files to the last commit: `git reset` 56
- TECHNIQUE 20 Deleting untracked files: `git clean` 57
- TECHNIQUE 21 Ignoring files: `.gitignore` 59
- TECHNIQUE 22 Deleting ignored files 60
- TECHNIQUE 23 Temporarily stashing some changes: `git stash` 61
- TECHNIQUE 24 Reapplying stashed changes: `git stash pop` 63
- TECHNIQUE 25 Clearing stashed changes: `git stash clear` 64
- TECHNIQUE 26 Assuming files are unchanged 64
- TECHNIQUE 27 Listing assumed-unchanged files 65
- TECHNIQUE 28 Stopping assuming files are unchanged 66
- 3.1 Summary 67

4 *History visualization* 68

- TECHNIQUE 29 Listing only certain commits 69
- 4.1 `git show` 70
 - TECHNIQUE 30 Listing commits with different formatting 71
- 4.2 Custom output format 74
- 4.3 Releasing logs: `git shortlog` 75
- 4.4 The ultimate log output 75
 - TECHNIQUE 31 Showing who last changed each line of a file: `git blame` 76
 - TECHNIQUE 32 Finding which commit caused a particular bug: `git bisect` 78
- 4.5 Automating `git bisect` 82
- 4.6 Summary 83

5 *Advanced branching* 84

- TECHNIQUE 33 Merging branches and always creating a merge commit 85
- 5.1 Merge strategies 87
 - TECHNIQUE 34 Resolving a merge conflict 88
- 5.2 Using a graphical merge tool 93
 - TECHNIQUE 35 Resolving each merge conflict only once: `git rerere` 94
 - TECHNIQUE 36 Creating a tag: `git tag` 95
 - TECHNIQUE 37 Generating a version number based on previous tags: `git describe` 97
 - TECHNIQUE 38 Adding a single commit to the current branch: `git cherry-pick` 98
 - TECHNIQUE 39 Reverting a previous commit: `git revert` 101
 - TECHNIQUE 40 Listing what branches contain a commit: `git cherry` 102
- 5.3 Summary 103

6 *Rewriting history and disaster recovery* 104

- TECHNIQUE 41 Listing all changes including history rewrites: `git reflog` 105
- 6.1 Avoiding and recovering from disasters 106
 - TECHNIQUE 42 Resetting a branch to a previous commit: `git reset` 107
 - TECHNIQUE 43 Rebasing commits on top of another branch: `git rebase` 110

TECHNIQUE 44	Rebasing commits interactively: git rebase --interactive	114
TECHNIQUE 45	Pulling a branch and rebasing commits: git pull --rebase	117
TECHNIQUE 46	Rewriting history on a remote branch: git push --force	119
TECHNIQUE 47	Rewriting the entire history of a branch: git filter-branch	121
6.2	Summary	125

PART 3 **ADVANCED GIT** **127**

7 *Personalizing Git* **129**

TECHNIQUE 48	Setting the configuration for all repositories	130
TECHNIQUE 49	Setting the configuration for a single repository	131
7.1	Useful configuration settings	133
	<i>Colored output in Git</i> 133 ■ <i>Git 2.0's push defaults</i> 133	
	<i>Pruning branches automatically</i> 134 ■ <i>Ignoring files across all</i>	
	<i>repositories: global ignore file</i> 134 ■ <i>Displaying help output in</i>	
	<i>your web browser</i> 135 ■ <i>Storing passwords in the OS X</i>	
	<i>keychain</i> 135 ■ <i>Storing arbitrary text in Git configuration</i> 136	
	<i>Autocorrecting misspelled commands</i> 136	
TECHNIQUE 50	Aliasing commands	137
7.2	Sharing your Git (or other) configuration between machines	138
TECHNIQUE 51	Showing the current branch in your terminal prompt	139
7.3	Summary	140

8 *Vendoring dependencies as submodules* **141**

8.1	When are submodules useful?	141
TECHNIQUE 52	Adding a git submodule: git submodule add	143
TECHNIQUE 53	Showing the status of submodules: git submodule status	146
TECHNIQUE 54	Updating and initializing all submodules: git submodule update --init	147
TECHNIQUE 55	Running a command in every submodule: git submodule foreach	149
8.2	Summary	150

9 *Working with Subversion* 151

- TECHNIQUE 56 Importing an SVN repository into a Git repository 151
- 9.1 Subversion branches and tags 154
- 9.2 Subversion ignore rules 155
- 9.3 Updating a Subversion repository 156
- 9.4 Subversion authors and committers 158
- 9.5 Viewing a Subversion repository in GitX 158
- 9.6 Migrating a Subversion repository to Git 158
 - TECHNIQUE 57 Committing and pushing to an SVN repository from a Git repository 159
- 9.7 Local branching and tagging 160
 - TECHNIQUE 58 Accessing a GitHub repository with Subversion 161
- 9.8 Summary 162

10 *GitHub pull requests* 163

- 10.1 What are pull requests and forks? 163
- 10.2 Interacting with GitHub from the command-line: gh 165
 - TECHNIQUE 59 Making a pull request in the same repository: gh pull-request 165
 - TECHNIQUE 60 Making a pull request from a forked repository: gh fork 168
 - TECHNIQUE 61 Merging a pull request from the same repository 169
 - TECHNIQUE 62 Merging a pull request from a forked repository: gh merge 172
- 10.3 Summary 173

11 *Hosting a repository* 174

- TECHNIQUE 63 Initializing a local repository in a server hosting format: git init --bare 175
- TECHNIQUE 64 Mirroring a repository: git clone --mirror 176
- TECHNIQUE 65 Sharing a repository with other users on the same network: git daemon 178
- TECHNIQUE 66 Displaying a repository in a browser: git instaweb 180
- 11.1 Advanced Git hosting 182
- 11.2 Summary 183

PART 4 GIT BEST PRACTICES 185

12 *Creating a clean history* 187

- 12.1 Writing a good commit message 187
- 12.2 Building a commit from parts of files: `git add --patch` 188
- 12.3 Graphically building a commit from parts of files 191
 - Graphically building a commit in GitX* 191 ■ *Graphically building a commit in Git Gui* 193
- 12.4 Avoiding whitespace issues: `git diff --check` 194
- 12.5 Summary 195

13 *Merging vs. rebasing* 196

- 13.1 CMake's workflow 196
 - Workflow commands* 198
- 13.2 Homebrew's workflow 200
 - Workflow commands* 201
- 13.3 CMake workflow pros and cons 202
- 13.4 Homebrew workflow pros and cons 203
- 13.5 Picking your strategy 204
- 13.6 Summary 205

14 *Recommended team workflows* 206

- 14.1 GitHub Flow 207
 - Pros and cons* 209
- 14.2 Git Flow 209
 - Pros and cons* 212
- 14.3 Mike Flow 213
 - Mike Flow Single* 213 ■ *Mike Flow Multiple* 215 ■ *Pros and cons* 217
- 14.4 Which workflow is for you? 218
- 14.5 Summary 219

appendix A Git installation 221

appendix B Creating a GitHub account and repository 224

appendix C Commented Git configuration 229

appendix D Why use version control? 234

index of Git methods 239

index 241

foreword

I first heard of Git probably nine years ago, around the end of 2005, right around when v1.0 was being tagged. Reading through my friend Mike's book, it amazes me just how much has changed in what I'm only now realizing has been nearly a decade of history.

At the time I was introduced to Git, it was still very much more of a collection of tools for content management and less a version control system. It was installed as hundreds of commands such as `git-update-index` instead of the single `git` command we now call. In fact, in my first exposure to it, I used it at work as a transport mechanism for media content, instead using Perforce to version-control the code we wrote around Git.

It fascinated me how clean and simple the data model was and how easy it was to manipulate to do anything you wanted, not simply what was prescribed and intended. Equally fascinating was how difficult it was to learn. It took my friend weeks to drill into me how this thing worked, but after that I was totally hooked and have spent most of the decade since helping people use it and understand it.

If you're still struggling with Git, this book does a great job of leading you through the basics in a straightforward manner. If you're already comfortable with those, parts 3 and 4 will help you see other ways you could be using Git to make your team's workflows simpler or more effective. Although I've spent a lot of time learning and explaining to others the elegant simplicity of the data model, I've never helped run a huge open source effort such as Homebrew as Mike has, and that experience is invaluable in explaining more advanced and complex workflows as Mike describes here.

I hope you enjoy this book, and I hope you use it to understand and enjoy the amazing benefits that a tool like Git can bring to your daily development life. I'm proud of Mike for taking on this challenge and for producing a great book on a tool that has made the transition over the past decade from being an interesting hobbyist's script to an essential part of millions of software developers' daily lives.

SCOTT CHACON
AUTHOR OF *Pro Git*

preface

As all good software engineers will agree: version control is important. I was first introduced to version control while studying computer science at university and dabbling with open source in my free time. CVS and Subversion started being used for managing our small pieces of coursework, group projects, and larger open source projects we interacted with. Initially it all felt a bit like overkill: why do I need this when I'm working by myself? It only took a few personal screw-ups for me to understand why version control is so important.

I was introduced to Git in 2006 while working on audio drivers for the Linux kernel at a summer internship. It was a fairly terrifying beast in those days: incredibly fast compared to its competitors, impressive in how it managed almost everything offline, but with an arcane user interface that was full of jargon. The jargon has decreased, but everything else remains fairly similar: Git is fast, impressive, and confusing.

My first few jobs after graduating found most companies still using Subversion. Git was one of those things used by kernel hackers and the particularly hardcore open source crowd, and it wasn't ready for mainstream use yet. I liked the glimpse I'd had of it, so I continued using it stealthily with `git svn`, a bridge that allows you to access Subversion repositories using Git. Eventually GitHub came along and convinced people that Git could be easy to use, and I was able to use Git more regularly. I became a maintainer of the Homebrew Mac OS X package manager, and more employers and clients began to use GitHub.

I think because I was a relatively early adopter of Git and the type of person who needs to know how things work, I ended up helping others with Git: first at my own workplace and then through training courses, talks, and helping others in the

open-source community. Eventually I was contacted by another publisher about writing a Git book, and I contacted Manning to ask if I could write with them. The result is *Git in Practice*, a book that takes you from intermediate- to advanced-level Git tools and workflows that I've found useful while working as a software engineer.

acknowledgments

There are many people without whom this book wouldn't be a reality. First, I'd like to thank Dan Maharry, my development editor at Manning, who kept me motivated and provided excellent feedback that has resulted in a far better book than I could have created on my own. Additionally, many others who work at and with Manning have helped me along the way, particularly Jay Bryant, Michael Stephens, Bert Bates, Benjamin Berg, Glenn Burnside, Tiffany Taylor, and Ozren Harlovic.

Thanks to Scott Chacon for his great book *Pro Git*, which helped me while I was learning Git, and for agreeing to write the foreword.

Thanks to Karsten Strøbæk for his technical proofread of the manuscript shortly before it went into production, to Gerd Koenig for his help, and to the following reviewers who provided invaluable feedback: Alan Lenton, Art Bergquist, Chris Couchoud, Clare Macrae, Corey Gallon, Gregor Zurowski, Jean-Pol Landrain, John Guthrie, Ken Fricklas, Nathan Davies, Patrick Toohey, Rick Umali, Tarin Gamberini, and Zarko Jovicic.

Thanks to Linus Torvalds, without whom this wonderful version control system would not exist, and to Tom Preston-Werner, Chris Wanstrath, and PJ Hyett for starting GitHub, without which Git wouldn't have nearly the widespread usage it does today. Also thanks to them for hiring me!

Finally, I'd like to thank my wife Lindsay for putting up with my occasional whining about writing this book and for her understanding about how much time I had to spend on it.

about this book

Git in Practice is not a comprehensive guide of every single command and feature available with Git. A book that covered this would be considerably longer than the one you find before you, and you'd learn a lot of unnecessary information. Instead, this book provides a detailed understanding of all the features of Git that are useful when you're using Git to manage source code for software projects. I decided on the contents by looking at my Terminal shell history and prioritizing commands based on how much I use them. I also discuss workflows and best practices used in the real world to organize software projects when using Git.

This book isn't written for every software developer but, as an *In Practice* book, aims at the intermediate to advanced level. To get the most out of this book, you should be able to relate to or exceed the knowledge of one of these target readers:

- Software developer who uses Git and knows enough to interact with a Git repository but lacks understanding of any features beyond those provided by other version control systems
- Software developer with advanced Subversion knowledge (able to branch, merge, and set/get Subversion metadata variables) who wishes to learn how to use Git
- Software developer who wishes to learn Git best practices to contribute to open source projects on GitHub
- Trainer with Git experience who wants to learn about more conceptual topics, commands they may have missed, and workflows, and who wants a resource to help coach others about using Git

- System administrator who has a good understanding of version control but wants to create Git repositories for scripts and/or host repositories for teams

If you don't know this much yet, don't worry. You can either try to work through this book and reread parts you struggle with or pick up a beginner's guide to Git through Manning, another publisher, or online and come back to *Git in Practice* afterward.

This book also has some recommended prerequisite knowledge and tools:

- *Command line*—I assume you're familiar with the command line, because Git is only used as a command-line tool in this book. For example, you need to be able to create, edit, and remove directories and files from the command line. This includes Windows users, who will need to install MSYSGit (which provides Git in a Unix shell on Windows).
- *Software development*—This book assumes a basic understanding of (and desire to learn more about) how version control is used by a team of software developers to build software projects.
- *Git or version control*—You should understand versions, branching, and merging in Git (such as `git commit`) or another version control system (such as Subversion `commit`, `log`, `copy`, `switch`, and `merge`).
- *GitHub*—You should be able to access a free GitHub account.
- *Programming languages*—You don't need to know any particular programming language.

Again, if you don't know any of this, don't worry. The book may be trickier for you, but it shouldn't be impossible to work through, and there's nothing that another beginner title or a search online won't be able to help you with.

Many of the chapters are organized into numbered Techniques, and these in turn consist of a Problem, Solution, and Discussion section for each Technique.

Roadmap

Part 1 (chapters 1 and 2) flies through the basics of using Git while teaching you the underlying concepts that are often misunderstood or omitted in beginners' guides. You may know how to use Git already, but I encourage you to persevere through this part anyway; it sets a good foundation that the rest of the book builds on:

- Chapter 1 covers why Git was created and how to create and interact with a local Git repository by committing files, viewing history, examining changes between commits, and using graphical tools.
- Chapter 2 discusses connecting your local Git repository to other repositories over the internet, allowing you to send and receive changes.

Part 2 (chapters 3–6) covers the most essential commands to learn when using Git either alone or with a team. This is where you'll start to see some of the more powerful differences between Git and other version control systems, with advanced branching capabilities and history rewriting:

- Chapter 3 discusses using interacting with files and directories in a Git repository, ignoring files or changes, deleting files based on their status in the repository, resetting files to a previous state, and temporarily storing changes to files to be reapplied later.
- Chapter 4 covers using Git's history-querying capabilities and configuring history to display the most useful information for any particular task.
- Chapter 5 examines using branches effectively with merges, dealing with conflicts, tagging, and reverting commits.
- Chapter 6 covers changing previous commits in a Git repository (known as rewriting history) and avoiding losing your work when doing so or through data corruption.

Part 3 (chapters 7–11) provides some bonus Git skills that will help you be more efficient. These chapters introduce a collection of techniques that won't be used with every Git project but may be important for new ones:

- Chapter 7 covers personalizing Git's configuration so you can work most effectively and efficiently given the current project, computer, or version of Git you're working on.
- Chapter 8 discusses using Git's submodules to reference other Git repositories from inside a Git repository. This is particularly useful when you're dealing with software dependencies.
- Chapter 9 covers interacting with Subversion repositories using Git either as a day-to-day workflow or as a one-time import and migration of a Subversion repository to Git.
- Chapter 10 examines using GitHub's pull request and forks features to work effectively in teams and contribute to open source software.
- Chapter 11 covers how Git repositories are hosted and migrated.

Part 4 (chapters 12–14) discusses different workflows and best practices for using Git as part of a team of software developers. You will be able to compare and contrast differing approaches and decide which is best for your team:

- Chapter 12 covers how to ensure that your commit history is clean, readable, and useful by using small, meaningful commits.
- Chapter 13 discusses the CMake and Homebrew open source projects' different workflows and compares them to examine the differences between rebase-heavy and merge-heavy workflows.
- Chapter 14 examines the most widely used workflows and my recommended workflows for working with Git as part of a software team, and how to manage branching, merging, rebasing, and releasing new versions of software.

The book also has four appendixes:

- Appendix A explains how to install Git on Windows, Mac, Linux, and Unix.

- Appendix B covers creating a GitHub account and repository for use throughout this book.
- Appendix C examines my Git configuration file: what I've customized, how, and why.
- Appendix D discusses why you should use version control and how to convince others that doing so is a good idea.

About the code

This book works through creating the Git repository for a silly example book called *Git in Practice: Redux*. This is available to download on GitHub at <https://github.com/GitInPractice/GitInPracticeRedux>. Note that the current state of this repository is the state after following all the examples in the book, so you should browse to the beginning of the history if you wish to follow along with the book's listings.

I also used a Git repository when writing this book in AsciiDoc files. If you wish to access this for an online copy of the book, to submit changes, or to see how it changed over time, please email me your GitHub username at mike+gitinpractice@mikemcquaid.com and I will grant you access to the repository.

My commented Git configuration is available in my dotfiles repository on GitHub at <https://github.com/mikemcquaid/dotfiles> and in appendix C.

The code is also available for download from the publisher's website at www.manning.com/GitinPractice.

Author Online

Purchase of *Git in Practice* includes free access to a private web forum run by Manning Publications where you can make comments about the book, ask technical questions, and receive help from the author and from other users. To access the forum and subscribe to it, point your web browser to www.manning.com/GitinPractice.

This page provides information on how to get on the forum once you're registered, what kind of help is available, and the rules of conduct on the forum. Manning's commitment to our readers is to provide a venue where a meaningful dialog between individual readers and between readers and the author can take place. It's not a commitment to any specific amount of participation on the part of the author, whose contribution to the forum remains voluntary (and unpaid). We suggest you try asking the author some challenging questions lest his interest stray!

The Author Online forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

About the author

Mike McQuaid works as a software engineer at GitHub, speaks at conferences, and trains people in using Git. He has contributed widely to Git-based open source software, including Qt and the Linux kernel, and is a maintainer of the Git-based Homebrew project, a popular OS X package manager.

about the cover illustration

The figure on the cover of *Git in Practice* is captioned “A Kamtchatdale in her full dress in 1768.” The Kamtchatka Peninsula is the eastern-most part of Russia, lying between the Pacific Ocean to the east and the Sea of Okhotsk to the west. The illustration is taken from Thomas Jefferys’ *A Collection of the Dresses of Different Nations, Ancient and Modern*, London, published between 1757 and 1772. The title page states that these are hand-colored copperplate engravings, heightened with gum arabic. Thomas Jefferys (1719–1771) was called “Geographer to King George III.” He was an English cartographer who was the leading map supplier of his day. He engraved and printed maps for government and other official bodies and produced a wide range of commercial maps and atlases, especially of North America. His work as a map maker sparked an interest in local dress customs of the lands he surveyed and mapped; they are brilliantly displayed in this four-volume collection.

Fascination with faraway lands and travel for pleasure were relatively new phenomena in the eighteenth century and collections such as this one were popular, introducing both the tourist as well as the armchair traveler to the inhabitants of other countries. The diversity of the drawings in Jefferys’ volumes speaks vividly of the uniqueness and individuality of the world’s nations centuries ago. Dress codes have changed, and the diversity by region and country, so rich at one time, has faded away. It is now often hard to tell the inhabitant of one continent from another. Perhaps, trying to view it optimistically, we have traded a cultural and visual diversity for a more varied personal life—or a more varied and interesting intellectual and technical life.

At a time when it is hard to tell one computer book from another, Manning celebrates the inventiveness and initiative of the computer business with book covers based on the rich diversity of national costumes three centuries ago, brought back to life by Jefferys’ pictures.

Part 1

Introduction to Git

Part 1 (chapters 1–2) will fly through the basics of using Git while teaching you the underlying concepts that are often misunderstood or omitted in beginners’ guides. You may know how to use Git already, but I encourage you to persevere through this part of the book; it’s a good foundation that the rest of the book will build on.

This part covers the following topics:

- Why Git was created
- How to create and commit files in a new local Git repository
- How to view the history of a Git repository at the command line or using graphical tools
- How to download changes from and upload changes to a remote Git repository
- How to create and merge branches

1

Local Git

This chapter covers

- Why Git was created
- Creating a new local Git repository
- Committing files into a Git repository
- Viewing the history of a Git repository
- Using gitk/GitX to visualize the history of a Git repository
- Viewing the differences between Git commits

In this chapter, you'll learn how and why to use Git as a local version control system. Let's start by learning why Git is so widely used by programmers.

1.1 Why do programmers use Git?

Git was created by a programmer to be used by programmers. Linus Torvalds, the creator of the Linux kernel, started writing Git in 2005 with the goal of having a distributed, open-source, high-performance, and hard-to-corrupt version control system for the Linux kernel project to use. Within a week, Git's source code was hosted inside the Git version control system; and within two and a half months, version 2.6.12 of the Linux kernel was released using Git.

Since its initial creation for the Linux kernel, Git has come to be used by many other open source projects, all sizes of companies, and large “social coding” Git hosting sites such as GitHub.

Git is my preferred method of software source code control. I also use it for versioning plain text files such as the manuscript for this book. Git has many strengths over other methods of source control. It stores all the history, branches, and commits locally. This means adding new versions or querying history doesn’t require a network connection. Git’s history-log viewing and branch creation are near-instant compared with, for example, Subversion’s, which are sluggish even on a fast network connection. Because Git stores changes locally, you’re not constrained by the work of others when working in a team. For example, remote branch updates and merges can be done independently so you can continue your edit/commit workflow without interruptions while still downloading and examining any changes made by others.

In Git, you can modify the history of branches and even rewrite commit data across the entire repository. It’s often useful to be able to make lots of small commits, which are later turned into a single commit, or to make commit messages contain more information after the fact, and Git’s history rewriting enables this. Even with this flexibility, every commit has a unique reference that survives rewriting or changes and won’t be purged until it’s missing from all branches for at least 30 days. This means it’s hard to accidentally lose work if it has been committed.

Git’s main downsides are that the command-line application’s interface is often counterintuitive: it makes frequent use of jargon that can only be adequately explained by understanding Git’s internals. Additionally, the official documentation can be hard to follow; it also uses jargon and has to detail the large number of options for Git’s commands. To the credit of the Git community, both the UI and documentation around Git have improved hugely over the years. This book will help you understand Git’s jargon and internal operations; these should help you to understand why Git does what it does when you run its Git commands.

Despite these downsides, the strengths of Git have proved too strong for many software projects to resist. Google, Microsoft, Twitter, LinkedIn, and Netflix all use Git, as do open source projects such as the Linux kernel (the first Git user), Perl, PostgreSQL, Android, Ruby on Rails, Qt, GNOME, KDE, Eclipse, and X.org.

Many of these projects and many users of Git have also been introduced to and use Git through a Git hosting provider. My favorite is GitHub, but various other paid and free alternatives are available.

1.2 Initial setup

Once you’ve installed Git version 1.8 or above (as detailed in appendix A), the first thing you need to do is to tell Git your name and email (particularly before creating any commits). Rather than usernames, Git uses a name and an email address to identify the author of a commit.

My name is Mike McQuaid and my email address is mike@mikemcquaid.com, so I would run this.

Listing 1.1 Setting a name and email address

```
# git config --global user.name "Mike McQuaid"
# git config --global user.email mike@mikemcquaid.com
# git config --global user.email
mike@mikemcquaid.com
```

← 1 A run command

← 2 No output

← 3 Command output

All command output listings in this book show results like these:

- 1 A run command prefixed with a #.
- 2 In this case there was no output, so this line has been left intentionally blank.
- 3 Command output following the run command. In this case, the output is the previously set value of `user.email` (my email address).

HOW CAN YOU FOLLOW THE LISTINGS? You can follow listings as you work through the book by running the listed commands (in a Terminal on Unix or Git Bash on Windows) and comparing the output to the listing. A POSIX-incompatible shell (which includes the Windows Command Prompt without Git Bash) may work for some listings but will fail for others that rely on POSIX syntax (such as the use of single quotes).

IS THERE AN EXAMPLE REPOSITORY? The steps followed in the examples in this book were used to create the demo repository `GitInPracticeRedux` on GitHub at <https://github.com/GitInPractice/GitInPracticeRedux>. Note that the current state of this repository is the state after following all the examples in the book, so you should browse to the beginning of the history if you wish to follow along with the listings.

Now Git is set up. To use it, you need to initialize a Git *repository* on your local machine.

Technique 1 Creating a repository: git init

A Git *repository* is the local collection of all the files related to a particular Git version control system and contains a `.git` subdirectory in its root. Git keeps track of the state of the files in the repository's directory on disk.

Git repositories store all their data on your local machine. Making commits, viewing history, and requesting differences between commits are all local operations that don't require a network connection. This makes all these operations much faster in Git than with centralized version control systems such as Subversion.

Typically you create a new repository by downloading another repository that already exists (known as *cloning* by Git and introduced in technique 8), but let's start by initializing an empty, new local repository.

Problem

You wish to create a new local Git repository in a new subdirectory named `GitInPracticeRedux`.

Solution

- 1 Change to the directory you wish to contain your new repository directory. For example: `cd /Users/mike/`.
- 2 Run `git init GitInPracticeRedux`.

Listing 1.2 Initializing a Git repository

```
# cd /Users/mike/
# git init GitInPracticeRedux    ← Requested name

Initialized empty Git repository in
/Users/mike/GitInPracticeRedux/.git/    ← Repository path
```

You’ve initialized a new local Git repository named `GitInPracticeRedux` accessible (on my machine) at `/Users/mike/GitInPracticeRedux`.

Discussion

We’re calling the repository `GitInPracticeRedux` rather than `GitInPractice` to differentiate it from the book itself.

WHERE CAN YOU SEE THE FULL SYNTAX REFERENCES FOR GIT COMMANDS? All `git` commands referenced in this book have complete references to all their possible syntax and arguments in Git’s help. This can be accessed for a given command by running the command suffixed with `--help`: for example, `git init --help`. This book will cover only the most common and useful commands and arguments.

`git init` can be run without any arguments to create the local Git repository in the current directory.

1.3 .git subdirectory

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

WHY IS THE .GIT DIRECTORY NOT VISIBLE? On some operating systems, directories starting with a period (.) such as `.git` are hidden by default. They can still be accessed in the console using their full path (such as `/Users/mike/GitInPracticeRedux/.git/`) but won’t show up in file listings in file browsers or by running a command like `ls /Users/mike/GitInPracticeRedux/`.

Let’s view the contents of the new Git repository by changing to the directory containing the Git repository and running the `find` command.

Listing 1.3 Listing files created in a new repository

```
# cd /Users/mike/ && find GitInPracticeRedux

GitInPracticeRedux/.git/config           ← ❶ Local configuration
GitInPracticeRedux/.git/description      ← ❷ Description file
GitInPracticeRedux/.git/HEAD             ← ❸ HEAD pointer
GitInPracticeRedux/.git/hooks/applypatch-msg.sample ← ❹ Event hooks
GitInPracticeRedux/.git/hooks/commit-msg.sample
GitInPracticeRedux/.git/hooks/post-update.sample
GitInPracticeRedux/.git/hooks/pre-applypatch.sample
GitInPracticeRedux/.git/hooks/pre-commit.sample
GitInPracticeRedux/.git/hooks/pre-push.sample
GitInPracticeRedux/.git/hooks/pre-rebase.sample
GitInPracticeRedux/.git/hooks/prepare-commit-msg.sample
GitInPracticeRedux/.git/hooks/update.sample
GitInPracticeRedux/.git/info/exclude      ← ❺ Excluded files
GitInPracticeRedux/.git/objects/info      ← ❻ Object information
GitInPracticeRedux/.git/objects/pack      ← ❼ Pack files
GitInPracticeRedux/.git/refs/heads        ← ❽ Branch pointers
GitInPracticeRedux/.git/refs/tags         ← ❾ Tag pointers
```

❶ contains the configuration of the local repository.

❷ is a file that describes the repository.

❸, ❽, and ❾ contain a HEAD pointer, branch pointers, and tag pointers, respectively, that point to commits.

❹ shows event hook samples (scripts that run on defined events). For example, *pre-commit* is run before every new commit is made.

❺ contains files that should be excluded from the repository.

❻ and ❼ contain object information and pack files, respectively, that are used for object storage and reference.

You shouldn't edit any of these files directly until you have a more advanced understanding of Git (or never). You'll instead modify these files and directories by interacting with the Git repository through Git's filesystem commands, introduced in chapter 3.

1.4 Creating a new commit: *git add*, *git commit*

To do anything useful in Git, you first need one or more commits in your repository. A *commit* is created from the changes to one or more files on disk. The typical workflow is that you'll change the contents of files in a repository, review the *diffs*, add them to the *index*, create a new commit from the contents of the index, and repeat this cycle.

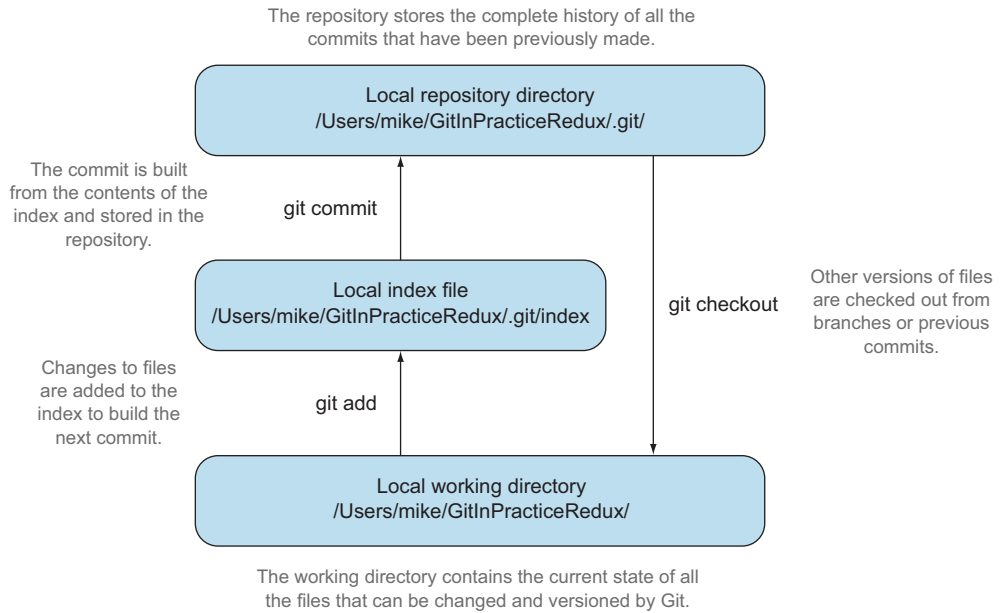


Figure 1.1 Git add/commit/checkout workflow

Git's index is a staging area used to build up new commits. Rather than require all changes in the working tree to make up the next commit, Git allows files to be added incrementally to the index. The add/commit/checkout workflow can be seen in figure 1.1.

Technique 2 *Building a new commit in the index staging area: git add*

Git doesn't add anything to the index without your instruction. As a result, the first thing you have to do with a file you want to include in a Git repository is request that Git add it to the index.

Problem

You wish to add an existing file `GitInPractice.asciidoc` to the index staging area for inclusion in the next commit.

Solution

- 1 Change directory to the Git repository, such as
`cd /Users/mike/GitInPracticeRedux/.`
- 2 Ensure that the file `GitInPractice.asciidoc` is in the current directory.
- 3 Run `git add GitInPractice.asciidoc`. There will be no output.

You've added `GitInPractice.asciidoc` to the index. If you were successful, the output of running `git status` should resemble the following.

Listing 1.4 Adding a file to the index

```
# git add GitInPractice.asciidoc
# git status
```

On branch master

← ① **Default branch output**

Initial commit

← ② **First commit**

Changes to be committed:

(use "git rm --cached <file>..." to unstage)

new file: GitInPractice.asciidoc

← ③ **New file in the index**

- ① is the first line of `git status` output. It shows the current *branch*, which, by default, is always *master*. Don't worry about creating branches for now; this will be covered later, in technique 11.
- ② shows that no commits have yet been made, and `git add` is being used to build the first commit.
- ③ shows the new file you've just added to the index (the staging area for the next commit).

Discussion

`git add` can also be passed directories as arguments instead of files. You can add everything in the current directory and its subdirectories by running `git add`.

When a file is added to the index, a file named `.git/index` is created (if it doesn't already exist). The added file contents and metadata are then added to the index file. You've requested two things of Git here:

- Git should track the contents of the file as it changes (this isn't done without an explicit `git add`).
- The contents of the file when `git add` was run should be added to the index, ready to create the next commit.

DOES GIT ADD NEED TO BE RUN MORE THAN ONCE? After you've added changes for `GitInPractice.asciidoc` to the index staging area with `git add`, they will be used in the next commit. After these changes to `GitInPractice.asciidoc` have been committed with `git commit` (introduced in technique 2), if you wish to add more changes to `GitInPractice.asciidoc` you'll need to run `git add GitInPractice.asciidoc` again. This is because, unlike in some version control systems, `git add` is used both to initially add a file to the Git repository and to request that changes to the file be used in the next commit.

Now that the contents of the file have been added to the index, you're ready to commit it.

Technique 3 Committing changes to files: `git commit`

Creating a *commit* stores the changes to one or more files. Each commit contains a message entered by the author, details of the commit author, a unique commit reference (in Git, *SHA-1 hashes* such as 86bb0d659a39c98808439fadb8dbd594bec0004d) a pointer to the preceding commit (known as the *parent commit*), the date the commit was created, and a pointer to the contents of files when the commit was made (see figure 1.2). The file contents are typically displayed as the *diff* (the differences between the files before and the files after the commit).

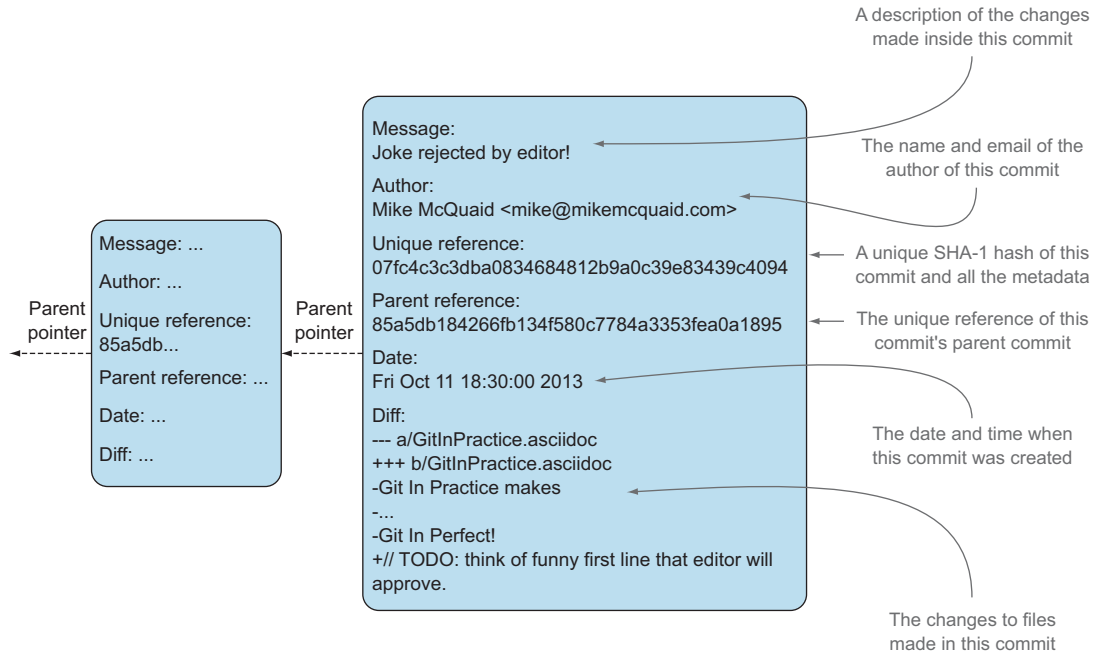


Figure 1.2 A typical commit broken down into its parts

WHY DO THE ARROWS POINT BACKWARD? As you may have noticed, figure 1.2 uses arrows pointing from commits to the previous commit. The reason is that commits contain a pointer to the *parent commit* and not the other way around; when a commit is made, it has no idea what the next commit will be.

Problem

You wish to commit the contents of an existing file `GitInPractice.asciidoc`, which has already been added to the index staging area. After this, you wish to make modifications to the file and commit them.

Solution

- 1 Change directory to the Git repository, such as `cd /Users/mike/GitInPracticeRedux/`.
- 2 Ensure that the file `GitInPractice.asciidoc` is in the current directory and that its changes were staged in the index with `git add`.
- 3 Run `git commit --message 'Initial commit of book.'` The output should resemble the following.

Listing 1.5 Committing changes staged in the index

```
# git commit --message 'Initial commit of book.'
[master (root-commit) 6576b68] Initial commit of book.
1 file changed, 2 insertions(+)
create mode 100644 GitInPractice.asciidoc
```

- 1 is the name of the branch where the commit was made (the default, `master`), the shortened SHA-1 (`6576b68`), and the commit message. (`root-commit`) means the same as the `Initial commit` you saw earlier. It's shown only for the first commit in a repository and means this commit has no parent.
- 2 shows the number of files changed and the number of lines inserted or deleted across all the files in this commit.
- 3 shows that a new file was created, along with the Unix file mode (`100644`). The file mode is related to Unix file permissions and the `chmod` command, but it isn't important in understanding how Git works and so can be safely ignored.

You've made a new commit containing `GitInPractice.asciidoc`.

WHAT IS A SHA-1 HASH? A *SHA-1 hash* is a secure hash digest function that is used extensively in Git. It outputs a 160-bit (20-byte) hash value, which is usually displayed as a 40-character hexadecimal string. The hash is used to uniquely identify commits by Git by their contents and metadata. They're used instead of incremental revision numbers (like in Subversion) due to the distributed nature of Git. When you commit locally, Git can't know whether your commit occurred before or after another commit on another machine, so it can't use ordered revision numbers. The full 40 characters are rather unwieldy, so Git often shows shortened SHA-1s (as long as they're unique in the repository). Anywhere that Git accepts a SHA-1 unique commit reference, it will also accept the shortened version (as long as the shortened version is still unique within the repository).

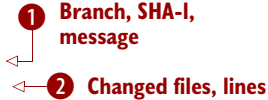
Let's create another commit:

- 1 Modify `GitInPractice.asciidoc`, and stage the changes in the index with `git add`.

- 2 Run `git commit --message 'Add opening joke. Funny?'`. The output should resemble the following.

Listing 1.6 Making a second commit

```
# git add GitInPractice.asciidoc
# git commit --message 'Add opening joke. Funny?'
[master 6b437c7] Add opening joke. Funny?
1 file changed, 3 insertions(+), 1 deletion(-)
```



1 Branch, SHA-1, message

2 Changed files, lines

- 1 has a different shortened SHA-1, because this is a new commit with different contents and metadata. No (root-commit) is shown, because this second commit has the first as its parent.
- 2 shows three insertions and one deletion because Git treats the modification of a line as the deletion of an old line and insertion of a new one.

You've made modifications to `GitInPractice.asciidoc` and committed them.

Discussion

The `--message` flag for `git commit` can be abbreviated to `-m` (all abbreviations use a single `-`). If this flag is omitted, Git opens a text editor (specified by the `EDITOR` or `GIT_EDITOR` environment variable) to prompt you for the commit message. These variables will also be used by other commands later in the book (such as interactive rebase, introduced in technique 44) when requesting text input.

`git commit` can be called with `--author` and `--date` flags to override the auto-set metadata in the new commit.

`git commit` can be called with a path (like `git add`) to do the equivalent of an add followed immediately by a commit. It can also take the `--all` (or `-a`) flag to add all changes to files tracked in the repository into a new commit. Although these methods all save time, they tend to result in larger (and therefore worse) commits, so I recommend avoiding their use until you've gotten used to using them separately. Technique 4 discusses several reasons small commits are better than large ones.

1.4.1 Object store

Git is a version control system built on top of an *object store*. Git creates and stores a collection of objects when you commit. The object store is stored inside the Git *repository*.

In figure 1.3, you can see the main Git objects we're concerned with: *commits*, *blobs*, and *trees*. There's also a *tag* object, but don't worry about tags until they're introduced in technique 36. Figure 1.2 showed an example of a commit object and how it stores metadata and referenced file contents. The file-contents reference is actually a reference to a *tree object*. A tree object stores a reference to all the *blob objects* at a particular point in time and other tree objects if there are any subdirectories. A blob object stores the contents of a particular version of a particular single file in the Git repository.

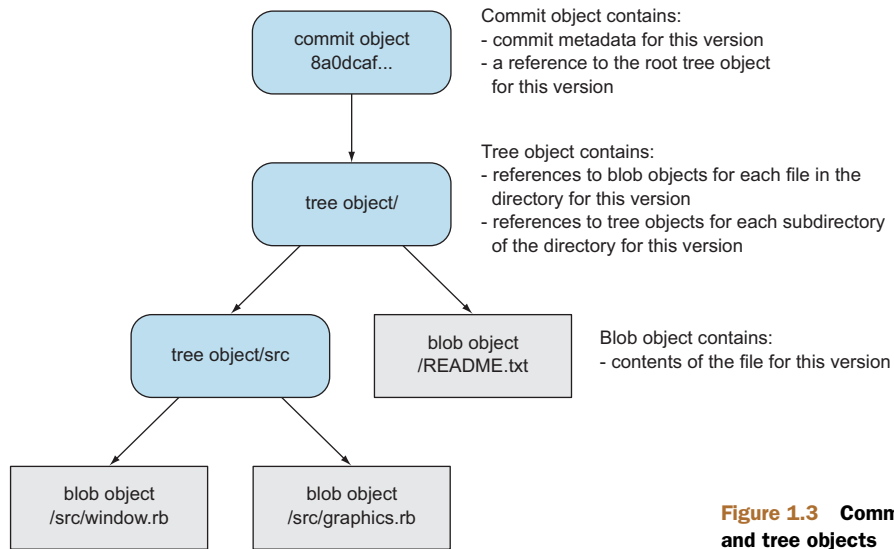


Figure 1.3 Commit, blob, and tree objects

SHOULD OBJECTS BE INTERACTED WITH DIRECTLY? When using Git, you should never need to interact with objects or object files directly. The terminology of *blobs* and *trees* isn't used regularly in Git or in this book, but it's useful to remember what these are so you can build a conceptual understanding of what Git is doing internally. When things go well, this should be unnecessary; but when you start to delve into more advanced Git functionality or Git spits out a baffling error message, then remembering *blobs* and *trees* may help you work out what has happened.

1.4.2 Parent commits

Every commit object points to its *parent commit*. The parent commit in a linear, branch-less history is the one that immediately preceded it. The only commit that lacks a parent commit is the *initial commit*: the first commit in the repository. By following the parent commit, its parent, its parent, and so on, you can always get back from the current commit to the initial commit. You can see an example of parent commit pointers in figure 1.4.

Now that you have two commits and have learned how they're stored, we can start looking at Git's history.

Technique 4 Viewing history: *git log*, *gitk*, *gitx*

The *history* in Git is the complete list of all commits made since the repository was created. The history also contains references to any *branches*, *merges*, and *tags* made within the repository. These three will be covered in technique 11, technique 14, and technique 36.

When you're using Git, you'll find yourself regularly checking the history: sometimes to remind yourself of your own work, sometimes to see why other changes were

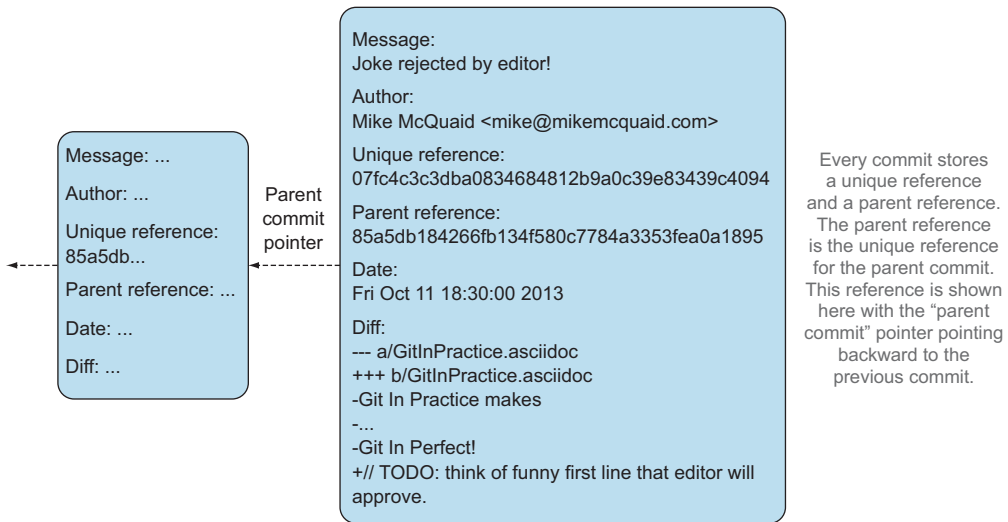


Figure 1.4 Parent commit pointers

made in the past, and sometimes reading new changes that have been made by others. In different situations, different pieces of data will be interesting, but all pieces of data are always available for every commit.

You may already have a sense for the fact that how useful the history is relies a great deal on the quality of the data entered into it. If I made a commit once per year with huge numbers of changes and a commit message of “fixes,” it would be fairly hard to use the history effectively. Ideally, commits are small and well-described; follow these two rules, and having a complete history becomes a very useful tool.

WHY ARE SMALL COMMITS BETTER? Sometimes it’s desirable to pick only some changed files (or even some changed lines within files) to include in a commit and leave the other changes to be added in a future commit. Commits should be kept as small as possible. This allows their message to describe a single change rather than multiple changes that are unrelated but were worked on at the same time. Small commits keep the history readable; it’s easier when looking at a small commit in the future to understand exactly why the change was made. If a small commit is later found to be undesirable, it can be easily reverted. This is much more difficult if many unrelated changes are clumped together into a single commit and you wish to revert a single change.

HOW SHOULD COMMIT MESSAGES BE FORMATTED? The commit message is structured like an email. The first line is treated as the subject and the rest as the body. The commit subject is used as a summary for that commit when only a single line of the commit message is shown, and it should be 50 characters or less. The remaining lines should be wrapped at 72 characters or less and separated from the subject by a single, blank line. The commit message should describe what the commit does in as much detail as is useful, in the present tense.

Let's learn how to view the history of a repository.

Problem

You wish to view the commit history (also known as the *log*) of a repository.

Solution

- 1 Change directory to the Git repository, such as `cd /Users/mike/GitInPracticeRedux/`.
- 2 Run `git log` and, if necessary, `q` to exit. The output should resemble the following.

Listing 1.7 History output

```
# git log

commit 6b437c7739d24e29c8ded318e683eca8f03a5260
Author: Mike McQuaid <mike@mikemcquaid.com>
Date:   Sun Sep 29 11:30:00 2013 +0100

    Add opening joke. Funny?

commit 6576b6803e947b29e7d3b4870477ae283409ba71
Author: Mike McQuaid <mike@mikemcquaid.com>
Date:   Sun Sep 29 10:30:00 2013 +0100

    Initial commit of book.
```

1 Unique SHA-1
2 Commit author
3 Committed date
4 Full commit message

The `git log` output lists all the commits that have been made on the current branch in reverse chronological order (the most recent commit comes first):

- 1 is the full 40-character commit reference.
- 2 shows the name and email address set by the person who made the commit.
- 3 shows the date and time when the commit was made.
- 4 shows the commit message subject on the first line; the remaining lines are the commit message body.

It's also useful to graphically visualize history. *Gitk* is a tool for viewing the history of Git repositories. It's usually installed with Git but may need to be installed by your package manager or separately. Its ability to graphically visualize Git's history is particularly helpful when history becomes more complex (say, with merges and remote branches). You can see it running on Windows 8.1 in figure 1.5.

There are more attractive, up-to-date, and platform-native alternatives to *gitk*. On Linux/Unix, I instead recommend using tools such as *gitg* for *gtk+*/*GNOME* integration and *QGit* for *Qt*/*KDE* integration. These can be installed using your package manager.

On OS X, there are tools such as *GitX* (and various forks of the project). OS X is my platform of choice, so I'll be using screenshots of the *GitX-dev* fork of *GitX* to discuss

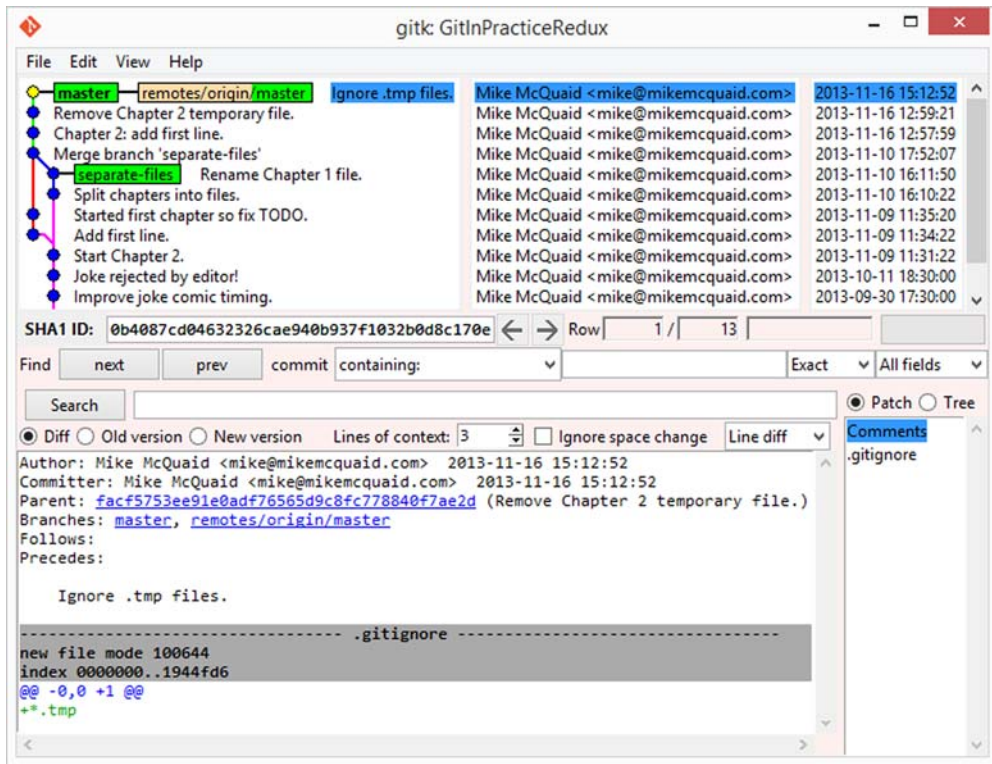


Figure 1.5 Gitk on Windows 8.1

history in this book, and I recommend you use it too if you use OS X. GitX-dev is available at <https://github.com/rowanj/gitx> and can be seen in figure 1.6.

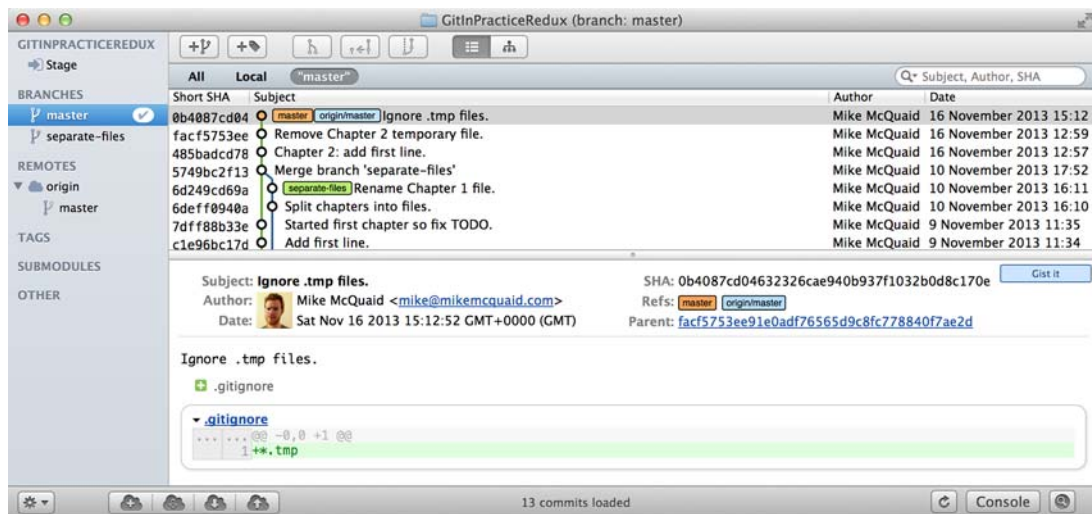


Figure 1.6 GitX-dev on OS X Mavericks

To view the commit history with gitk or GitX, follow these steps:

- 1 Change directory to the Git repository, such as `cd /Users/mike/GitInPracticeRedux/`.
- 2 Run `gitk` or `gitx`.

The GitX history (shown in figure 1.7) shows output similar to that of `git log` but in a different format. You can also see the current branch and the contents of the current commit, including the diff and parent SHA-1. There's a lot of information that doesn't differ between commits, however.

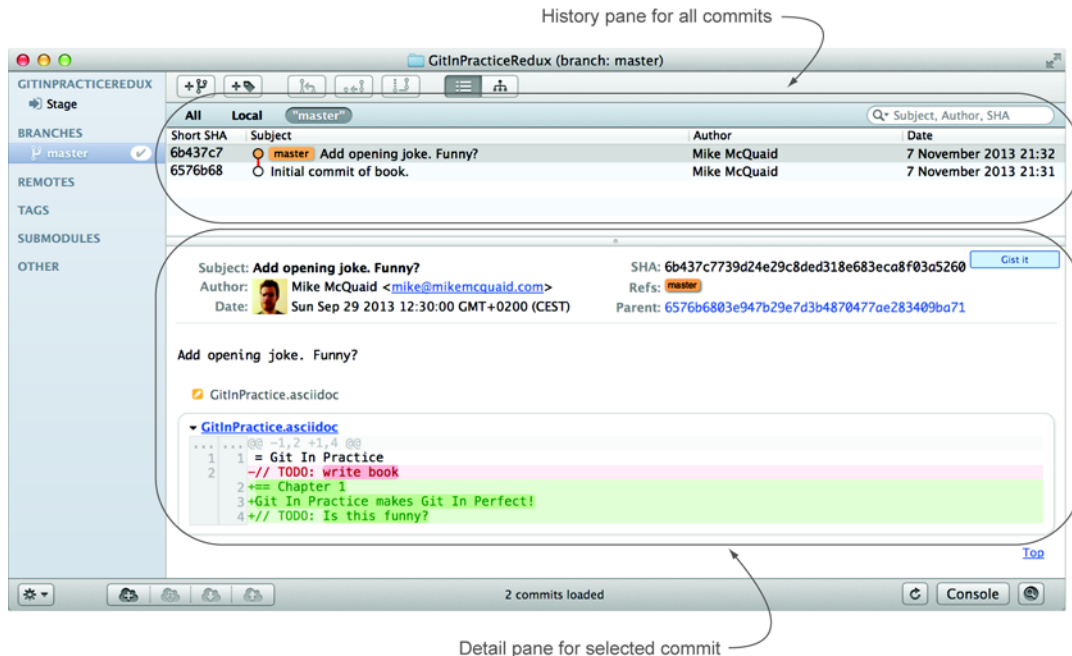


Figure 1.7 GitX history output

In figure 1.8, you can see the GitX history graph output. This format is used throughout the book to show the current state of the repository and/or the previous few commits. It concisely shows the unique SHA-1, all branches (only `master` in this case), the current local branch (shown in the GUI with an orange label), the commit message subject (the first line of the commit message), and the commit's author, date, and time.

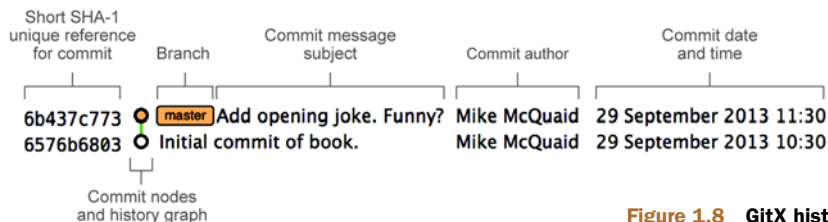


Figure 1.8 GitX history graph output

Discussion

`git log` can take revision or path arguments to specify the output history be shown starting at the given revision or only include changes to the requested paths. `git log` can take a `--patch` (or `-p`) flag to show the *diff* for each commit output. It can also take `--stat` or `--word-diff` flag to show a *diffstat* or *word diff*. These terms will be explained in technique 5.

1.5 Rewriting history

Git is unusual compared to many other version control systems in that it allows history to be rewritten. This may seem surprising or worrying; after all, didn't I just tell you that the history contains the entire list of changes to the project over time? Sometimes you may want to highlight only broader changes to files in a version control system over a period of time instead of sharing every single change that was made in reaching the final state.

Figure 1.9 shows a fairly common use case for rewriting history with Git. If you were working on some window code all morning and wanted your coworkers to see it later (or wanted to include it in the project), there's no need for everyone to see the mistakes you made along the way. In the figure the commits are *squashed* together: so instead of having three commits with the latter two fixing mistakes from the first commit, we've squashed them together to create a single commit for the window feature. You'd only rewrite history like this if you were working on a separate branch that didn't have other work from other people relying on it yet, because it has changed some parent commits (so, without intervention, other people's commits may point to commits that no longer exist). Don't worry too much about squashing work for now; just remember this as a situation where you may want to rewrite history. Much later, in technique 42, you'll learn how to rewrite history and the cases where it's useful and safe to do so.

What we're generally interested in when reading the history (and why we clean it up) is ensuring that the changes between commits are relevant (for example, don't make changes only to revert them immediately in the next commit five minutes later), minimal, and readable. These changes are known as *diffs*.

The history can give you a quick overview of all the previous commits. But querying the differences between any two arbitrary commits can also sometimes be useful, so let's learn how to do that.

Technique 5 Viewing the differences between commits: `git diff`

A *diff* (also known as a *change* or *delta*) is the difference between two commits. In Git you can request a diff between any two commits, branches, or tags. It's often useful to be able to request the difference between two parts of the history for analysis. For example, if an unexpected part of the software has recently started misbehaving, you can go back into the history to verify that it previously worked. If it did work previously, then you may want to examine the diff between the code in the different parts of the

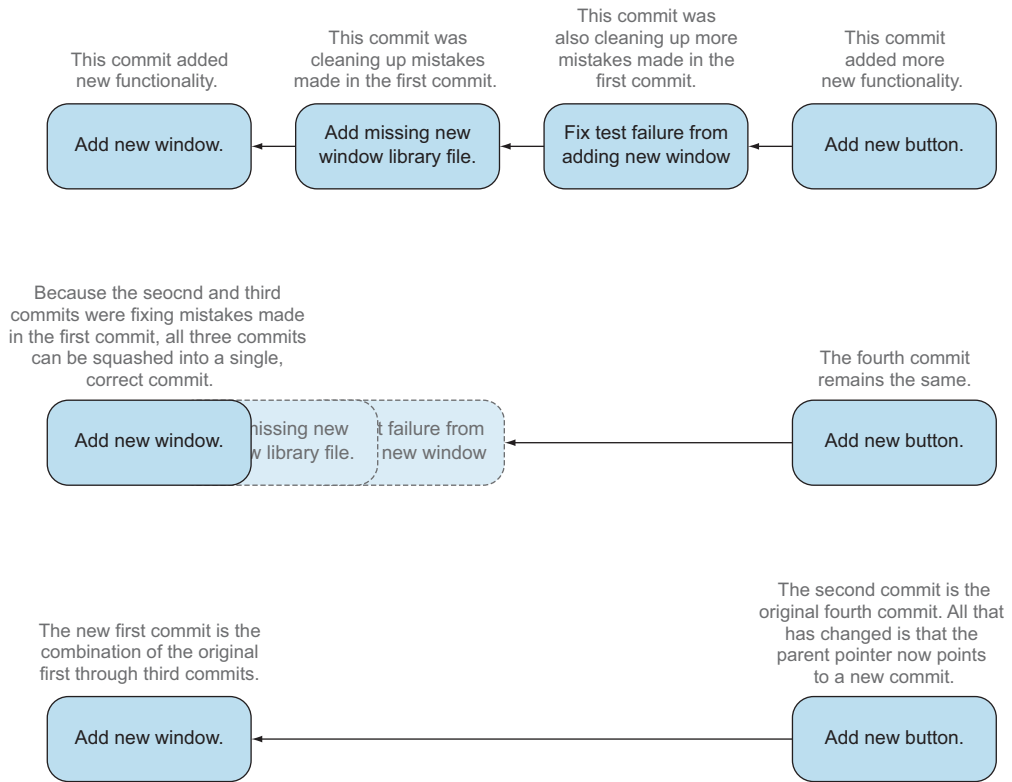


Figure 1.9 Squashing multiple commits into a single commit

history to see what has changed. The various ways of displaying diffs in version control typically allow you to narrow them down to the file, directory, and even committer.

Problem

You wish to view the differences between the previous commit and the latest.

Solution

- 1 Change directory to the Git repository, such as `cd /Users/mike/GitInPracticeRedux/`.
- 2 Run `git diff master~1 master` (you may need to press `Q` to exit afterward). The output should resemble the following.

Listing 1.8 Differences between the previous commit and the latest

```
# git diff master~1 master
```

← 1 git diff command

```
diff --git a/GitInPractice.asciidoc b/GitInPractice.asciidoc
index 48f7a8a..b14909f 100644
```

← 2 Virtual diff command

← 3 Index SHA-1 changes

```

--- a/GitInPractice.asciidoc      ← ④ Old virtual path
+++ b/GitInPractice.asciidoc      ← ⑤ New virtual path
@@ -1,2 +1,4 @@                   ← ⑥ Diff offsets
   = Git In Practice
-// TODO: write book               ← ⑦ Modified/deleted line
+== Chapter 1                      ← ⑧ Modified/inserted line
+Git In Practice makes Git In Perfect! ← ⑨ Inserted line
+// TODO: Is this funny?

```

- ① is the command that requests that Git show the diff between the commit before the top of master (master~1) and the commit on top of master. Both master~1 and master are *refs* and will be explained in section 1.7.
- ② is the invocation of the Unix `diff` command that Git is simulating. Git pretends that it's diffing the contents of two directories ④ and ⑤, and ② represents that. The `--git` flag can be ignored, because it just shows this is the Git simulation and the Unix `diff` command is never run.
- ③ shows the difference in the contents of the working tree between these commits. This can be safely ignored, other than noticing that these SHA-1s don't refer to the commits themselves.
- ④ is the simulated directory for the master~1 commit.
- ⑤ is the simulated directory for the master commit.
- ⑥ can be ignored. These are used by the Unix `diff` command to identify what lines the diff relates to for files that are too large to be shown in their entirety.
- ⑦ shows the previous version of a line that differs between the commits. Recall that a modified line is shown as a deletion and insertion.
- ⑧ shows the new version of a line that differs between the commits.
- ⑨ shows a new line that was added in the latter commit.

Discussion

`git diff` can take path arguments after a `--` to request only the differences between particular paths. For example, `git diff master~1 master -- GitInPractice.asciidoc` shows only the differences to the `GitInPractice.asciidoc` file between the previous and latest commits.

`git diff` without an argument displays the differences between the current working directory and the index staging area. `git diff master` displays the differences between the current working directory and the last commit on the default master branch.

If `git diff` is run with no arguments, it shows the differences between the index staging area and the current state of the files tracked by Git: any changes you've made but not yet added with `git add`.

1.6 Diff formats

Diffs are shown by default in Git (and in the previous example) in a format known as a *unified format diff*. Diffs are used often by Git to indicate changes to files, for example when navigating through history or viewing what you're about to commit.

Sometimes it's desirable to display diffs in different formats. Two common alternatives to a typical unified format diff are a *diffstat* and *word diff*.

Listing 1.9 Diffstat format

```
# git diff --stat master~1 master
GitInPractice.asciidoc | 4 +++-
1 file changed, 3 insertion(+), 1 deletions(-)
```

1 One file's changes
2 All files' changes

1 shows the filename that has been changed, the number of lines changed in that file, and +/- characters summarizing the overall changes to the file. If multiple files were changed, this would show multiple filenames, and each would have the lines changed for that file and +/- characters.

2 lists a summary of totals of the number of files changes and lines inserted/deleted across all files.

This diffstat shows the same changes as the unified format diff in the previous solution. Rather than showing the breakdown of exactly what has changed, it indicates what files have changed and a brief overview of how many lines were involved in the changes. This can be useful when getting a quick overview of what has changed without needing all the detail of a normal unified format diff.

Listing 1.10 Word diff format

```
# git diff --word-diff master~1 master

diff --git a/GitInPractice.asciidoc b/GitInPractice.asciidoc
index 48f7a8a..b14909f 100644
--- a/GitInPractice.asciidoc
+++ b/GitInPractice.asciidoc
@@ -1,2 +1,4 @@
= Git In Practice
{+= Chapter 1+}
{+Git In Practice makes Git In Perfect!+}
// TODO: [-write book-]{+Is this funny?+}
```

1 Added line
2 Modified line

1 shows a completely new line that was inserted, surrounded by {+}.

2 shows some characters that were deleted, surrounded by [-]; and some lines that were inserted, surrounded by {+}.

This word diff shows the same changes as the unified format diff in the previous solution. A word diff is similar to a unified format diff but shows modifications per word rather than per line. This is particularly useful when viewing changes to plain text

rather than code; in README files, we probably care more about individual word choices than knowing that an entire line has changed, and the special characters (`[-]{+}`) aren't used as often in prose as in code.

1.7 Refs

In Git, *refs* are the possible ways of addressing individual commits. They're an easier way to refer to a specific commit or branch when specifying an argument to a Git command.

The first ref you've already seen is a branch (which is `master` by default if you haven't created any other branches). Branches are pointers to specific commits. Referencing the branch name `master` is the same as referencing the SHA-1 of the commit at the top of the `master` branch, such as the short SHA-1 `6b437c7` in the last example. Whenever you might type `6b437c7` in a command, you can instead type `master`, and vice versa. Using branch names is quicker and easier to remember for referencing commits than always using SHA-1s.

Refs can also have modifiers appended. Suffixing a ref with `~1` is the same as saying "one commit before that ref." For example, `master~1` is the penultimate commit on the `master` branch: the short SHA-1 `6576b68` in the last example. Another equivalent syntax is `master^`, which is the same as `master~1` (and `master^^` is equivalent to `master~2`).

The second ref is the string `HEAD`. `HEAD` always points to the top of whatever you currently have checked out, so it's almost always the top commit of the current branch you're on. If you have the `master` branch checked out, then `master` and `HEAD` (and `6b437c7` in the last example) are equivalent. See the `master/HEAD` pointers demonstrated in figure 1.10.

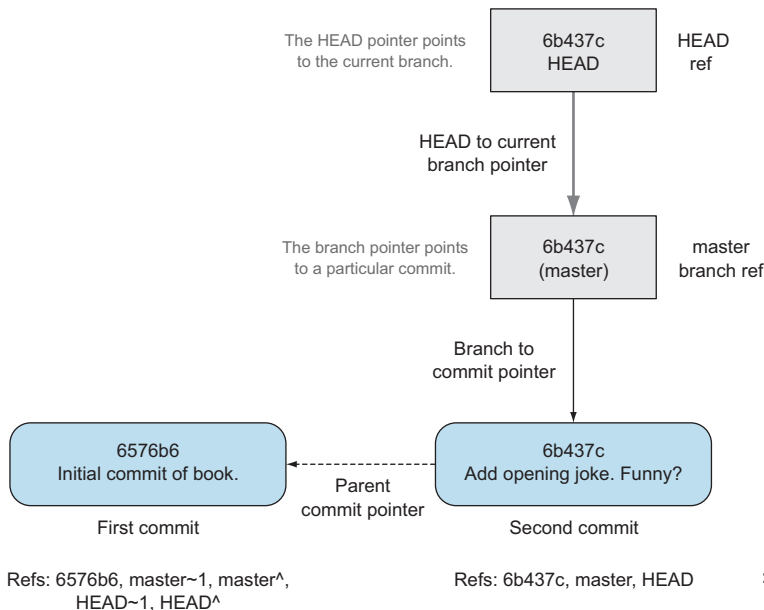


Figure 1.10 HEAD, master, and modified refs

These `git diff` invocations are all equivalent:

- `git diff master~1 master`
- `git diff master~1..master`
- `git diff master~1..`
- `git diff master^ master`
- `git diff master~1 HEAD`
- `git diff 6576b68 6b437c7`

You can also use the tool `git rev-parse` if you want to see what SHA-1 a given ref expands to.

Listing 1.11 Parsing refs

```
# git rev-parse master
6b437c77739d24e29c8ded318e683eca8f03a5260

# git rev-parse 6b437c7
6b437c77739d24e29c8ded318e683eca8f03a5260
```

There are more types of refs, such as remote branches and tags, but you don't need to worry about them now. They'll be introduced in technique 6 and technique 36.

1.8 Summary

In this chapter, you learned the following:

- Why Git is a good and high-performance version control system
- How to create a new local repository using `git init`
- How to add files to Git's index staging area using `git add`
- How to commit files to the Git repository using `git commit`
- How to view history using `git log` and `gitk/gitx`
- How to see the differences between commits using `git diff`
- How to use refs to reference commits

Remote Git



This chapter covers

- Downloading a remote repository
- Sending changes to and receiving changes from a remote repository
- Creating and receiving branches
- Merging commits from one branch to another

As you learned in technique 1, it's possible to work entirely with Git as a local version control system and never share changes with others. Usually, however, if you're using a version control system, you'll want to share changes: from simply sending files to a remote server for backup to collaborating as part of a large development team. Team collaboration also requires knowledge of how to create and interact with branches for working on different features in parallel. Let's start by adding a remote repository.

Technique 6 Adding a remote repository: `git remote add`

Typically when using version control, you'll want to share your commits with other people using other computers. With a traditional *centralized version control system* (such as Subversion or CVS), the repository is usually stored on another machine. As you make a commit, it's sent over the network, checked to see whether it can

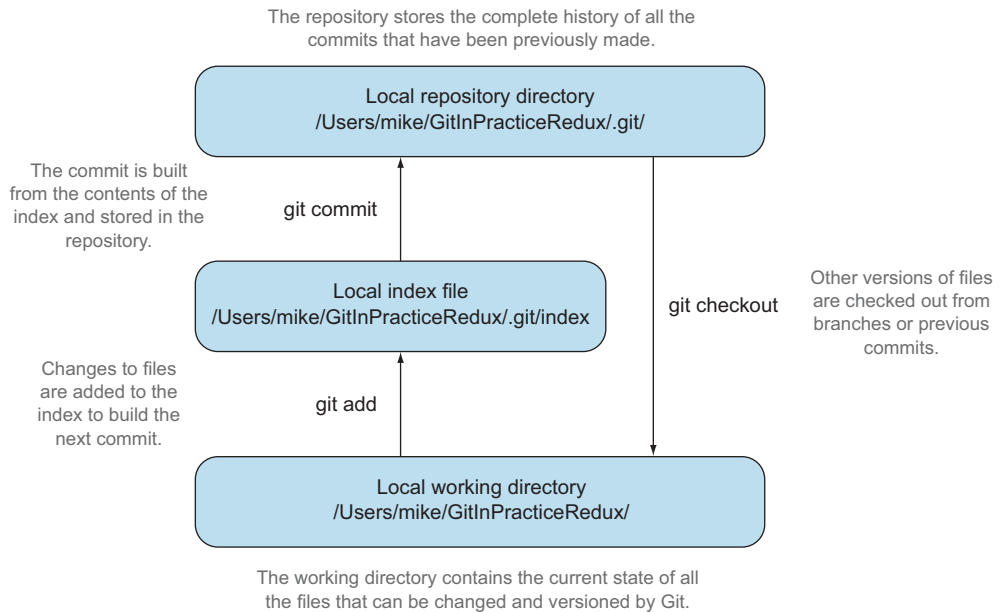


Figure 2.1 Git add/commit/checkout cycle

apply (there may be other changes since you last checked), and then committed to the version control system where others can see it.

With a *distributed version control system* like Git, every user has a complete repository on their own computer. Although there may be a centralized repository to which people send their commits, it isn't accessed unless specifically requested. All commits, branches, and history are stored offline unless users choose to send or receive commits from another repository.

Figure 2.1 shows the local Git cycle used in section 1.4. Files in the local working directory are modified and added with `git add` to the index staging area. The contents of the index staging area are committed with `git commit` to form a new commit, which is stored in the local repository directory. Later, this repository can be queried to view the differences between versions of files using `git diff`. In technique 12, you'll also see how to use `git checkout` to change to different local branches' versions of files.

Figure 2.2 shows the remote Git cycle we'll look at in this chapter. As in the local workflow, files are modified, are added, are committed, and can be checked out. But there are now two repositories: a local repository and a *remote repository*.

If your local repository needs to send data to or receive data from a repository on another machine, it will need to add a *remote repository*. A remote repository is one that's typically stored on another computer. `git push` sends your new commits to it, and `git fetch` retrieves from it any new commits made by others.

In technique 1, you created a local repository on your machine. Please sign up for a GitHub account and create a *remote repository* on GitHub (detailed in appendix B).

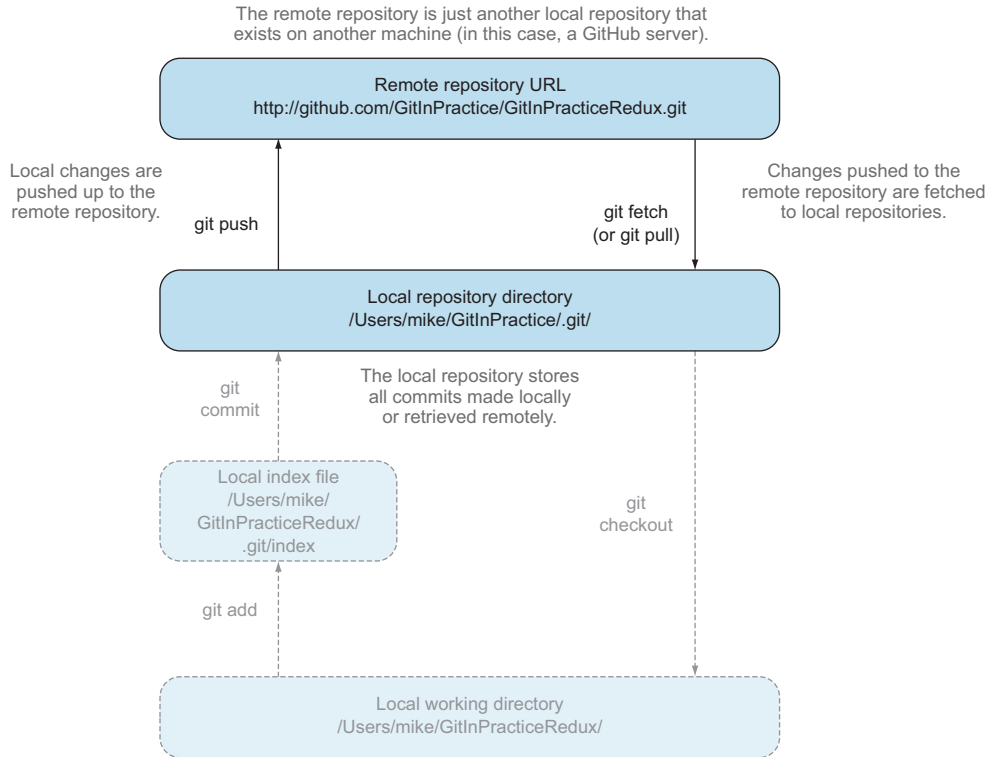


Figure 2.2 Git add/commit/push/pull/checkout cycle

You can use another Git hosting provider, but this book will assume the use of GitHub (because it's the most widely used).

The first action you're concerned with is adding to your previous local repository a reference for your newly created remote repository (also known as a *remote*) on GitHub so you can push and fetch commits.

Problem

You wish to add the new `GitInPractice` remote repository to your current repository.

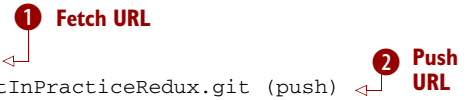
Solution

- 1 Change directory to the Git repository; for example, `cd /Users/mike/GitInPracticeRedux/`.
- 2 Run `git remote add origin` with your repository URL appended. So if your username is `GitInPractice` and your repository is named `GitInPracticeRedux`, run `git remote add origin https://github.com/GitInPractice/GitInPracticeRedux.git`. There will be no output.

You can verify that this remote has been created successfully by running `git remote --verbose`. The output should resemble the following.

Listing 2.1 Output: creating a remote repository

```
# git remote --verbose
origin https://github.com/GitInPractice/
      GitInPracticeRedux.git (fetch)
origin https://github.com/GitInPractice/GitInPracticeRedux.git (push)
```



❶ specifies the URL that `git fetch` uses to fetch new remote commits.

❷ specifies the URL that `git push` uses to send new local commits.

WHAT HAPPENS WHEN THE FETCH AND PUSH URLS DIFFER? The fetch and push URLs won't differ unless they've been set to do so with the `git remote` command or by Git configuration. It's almost never necessary to do this, so I won't cover it in this book.

You've added a remote named `origin` that points to the remote `GitInPracticeRedux` repository belonging to the `GitInPractice` user on GitHub. You can now send and receive changes from this remote. Nothing has been sent or received yet; the new remote is effectively just a named URL pointing to the remote repository location.

Discussion

`git remote` can also be called with the `rename` and `remove` (or `rm`) subcommands to alter remotes accordingly. `git remote show` queries and shows verbose information about the given remote. `git remote prune` deletes any remote references to branches that have been deleted from the remote repository by other users. Don't worry about this for now; remote branches will be covered in technique 13.

WHAT IS THE DEFAULT NAME FOR A REMOTE? You can have multiple remote repositories connected to your local repository, so the remote repositories are named. Typically, if you have a single remote repository, it's named `origin`.

2.1 Authoritative version storage

With centralized version control systems, the central server always stores the authoritative version of the code. Clients to this repository typically only store a small proportion of the history and require access to the server to perform most tasks. With a distributed version control system like Git, every local repository has a complete copy of the data. Which repository stores the authoritative version in this case? It turns out that this is merely a matter of convention; Git itself doesn't deem any particular repository to have any higher priority than another. Typically in organizations there is a central location (as with a centralized version control) that is treated as the authoritative version, and people are encouraged to push their commits and branches there.

The lack of authority for a particular repository with distributed version control systems is sometimes seen as a liability but can be a strength. The Linux kernel project (for which Git was originally created) uses this to provide a network of trust and a more manageable way of merging changes. When Linus Torvalds, the self-named “benevolent dictator” of the project, tags a new release, this is generally considered a new release of Linux. What’s in his repository (well, his publicly accessible one; he has multiple repositories between various personal machines that he doesn’t make publicly accessible) is generally considered to be what’s in Linux. Linus has trusted lieutenants from whom he can pull and merge commits and branches. Rather than every single merge to Linux needing to be done by Linus, he can leave some of it to his lieutenants (who leave some to their sub-lieutenants, and so on), so everyone only needs to worry about verifying and including the work of a small number of others. This particular workflow may not make sense in many organizations, but it demonstrates how distributed version control systems can allow different ways of managing merges to centralized version control.

Technique 7 *Pushing changes to a remote repository: git push*

You’ll eventually wish to send commits made in the local repository to a remote. To do this always requires an explicit action. Only changes specifically requested are sent, and Git (which can operate over HTTP, SSH, or its own protocol [git://]) ensures that only the differences between the repositories are sent. As a result, you can push small changes from a large local repository to a large remote repository very quickly as long as they have most commits in common.

Let’s push the changes you made in the repository in technique 3 to the newly created remote you made in technique 6.

Problem

You wish to push the changes from the local `GitInPracticeRedux` repository to the origin remote on GitHub.

Solution

- 1 Change directory to the Git repository: for example, `cd /Users/mike/GitInPracticeRedux/`.
- 2 Run `git push --set-upstream origin master`, and enter your GitHub username and password when requested. The output should resemble the following.

Listing 2.2 Output: pushing and setting an upstream branch

```
# git push --set-upstream origin master
```

```
Username for 'https://github.com': GitInPractice
```

```
Password for 'https://GitInPractice@github.com':
```

```
Counting objects: 6, done.
```

1 Username entry

2 Password entry

3 Object preparation and transmission

```

Delta compression using up to 8 threads.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (6/6), 602 bytes | 0 bytes/s, done.
Total 6 (delta 0), reused 0 (delta 0)
To https://github.com/GitInPractice/GitInPracticeRedux.git
 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.

```

Remote URL 4

5 Local/remote branch

Set tracking branch 6

- 1 and 2 are for your GitHub account. They may only be asked for the first time you push to a repository, depending on your operating system of choice (which may decide to save the password for you). They're always required to push to repositories but are only required for fetch when fetching from private repositories.
- 3 can be safely ignored in this and future listings; it's Git communicating details on how the files are being sent to the remote repository and isn't worth understanding beyond basic progress feedback.
- 4 matches the push URL from the `git remote --verbose` output earlier. It's where Git has sent the local commits.
- 5 indicates that this was a new branch on the remote. This is because the remote repository on GitHub was empty until you pushed this; it had no commits and thus no master branch yet. This was created by `git push. master -> master` indicates that the local master branch (the first of the two) has been pushed to the remote master branch (the second of the two). This may seem redundant, but it's shown here because it's possible (but ill-advised due to the obvious confusion it causes) to have local and remote branches with different names. Don't worry about local or remote branches for now; they will be covered starting in technique 11.
- 6 is shown because the `--set-upstream` option was passed to `git push`. By passing this option, you tell Git that you want the local master branch you've just pushed to *track* the origin remote's branch master. The master branch on the origin remote (which is often abbreviated `origin/master`) is now known as the *tracking branch* (or *upstream*) for your local master branch.

You have pushed your master branch's changes to the origin remote's master branch.

Discussion

The `git push --set-upstream` (or `-u`) flag and explicit specification of origin and master are only required the first time you push to create a remote branch (without them, some versions of Git may output fatal: The current branch master has no upstream branch.). After that, a `git push` with no arguments will default to running the equivalent of `git push origin master`. This is set up by default by `git clone` when you clone a repository.

```

6b437c77   master origin/master Add opening joke. Funny? Mike McQuaid 29 September 2013 11:30
6576b680   Initial commit of book. Mike McQuaid 29 September 2013 10:30

```

Figure 2.3 Local repository after `git push`

`git push` can take an `--all` flag, which pushes all branches and tags (introduced later in technique 36) at once. Be careful when doing this: you may push some branches with work in progress.

`git push` can take a `--force` flag, which disables some checks on the remote repository to allow rewriting of history. *This is very dangerous. Don't use this flag until after later reading (and re-reading) technique 46.*

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

Figure 2.3 shows the state of the repository after the `git push`. There's one addition since you last looked at it in section 1.7: the `origin/master` label. This is attached to the commit that matches the currently known state of the `origin` remote's `master` branch.

Figure 2.4 shows the remote repository on GitHub after the `git push`. The latest commit SHA-1 there matches your current latest commit on the `master` branch shown in figure 2.3 (although they're different lengths; remember, SHA-1s can always be shortened, as long as they remain unique). To update this in the future, you'd run `git push` again to push any local changes to GitHub.

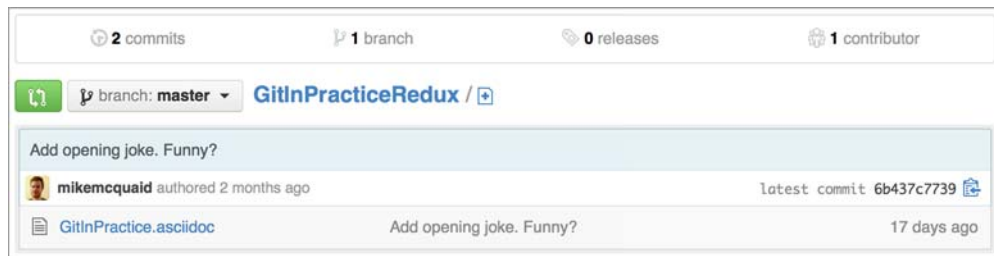


Figure 2.4 GitHub repository after `git push`

Technique 8 Cloning a remote/GitHub repository onto your local machine: `git clone`

It's useful to learn how to create a new Git repository locally and push it to GitHub. But you'll usually be downloading an existing repository to use as your local repository. This process of creating a new local repository from an existing remote repository is known as *cloning* a repository.

Some other version control systems (such as Subversion) use the terminology of *checking out* a repository. The reasoning for this is that Subversion is a centralized version control system, so when you download a repository locally, you're only downloading the latest revision from the repository. With Git, it's known as *cloning* because you're making a complete copy of that repository by downloading all commits, branches, and tags (introduced later in technique 36); you're putting the complete history of the repository onto your local machine.

You just pushed the entire contents of the local repository to GitHub, so let's remove the local repository and re-create it by cloning the repository on GitHub.

Problem

You wish to remove the existing `GitInPracticeRedux` local repository and re-create it by cloning from GitHub.

Solution

- 1 Change to the directory where you want the new `GitInPracticeRedux` repository to be created—say, `cd /Users/mike/` to create the new local repository in `/Users/mike/GitInPracticeRedux`.
- 2 Run `rm -rf GitInPracticeRedux` to remove the existing `GitInPracticeRedux` repository.
- 3 Run `git clone` with your repository URL appended. So if your username is `Git-InPractice` and your repository is named `GitInPracticeRedux`, run `git clone https://github.com/GitInPractice/GitInPracticeRedux.git`. The output should resemble the following.

Listing 2.3 Output: cloning a remote repository

```
# git clone https://github.com/GitInPractice/GitInPracticeRedux.git
Cloning into 'GitInPracticeRedux'...
remote: Counting objects: 6, done.
remote: Compressing objects: 100% (5/5), done.
remote: Total 6 (delta 0), reused 6 (delta 0)
Unpacking objects: 100% (6/6), done.
Checking connectivity... done
```

← ① Destination directory
← ② Object preparation and transmission

- ① is the directory in which the new `GitInPracticeRedux` local repository was created.
② can be safely ignored again (although if you're wondering why there were six objects, remember the different objects in the object store in technique 3).

You've cloned the `GitInPracticeRedux` remote repository and created a new local repository containing all its commits in `/Users/mike/GitInPracticeRedux`.

You can verify that this remote has been created successfully by running `git remote --verbose`. The output should resemble the following.

Listing 2.4 Output: remote repository

```
# git remote --verbose
origin https://github.com/GitInPractice/GitInPracticeRedux.git (fetch)
origin https://github.com/GitInPractice/GitInPracticeRedux.git (push)
```

Fetch URL
Push URL

Discussion

`git clone` can take `--bare` and `--mirror` flags, which create a repository suitable for hosting on a server. This will be covered more in chapter 13.

`git clone` can also take a `--depth` flag followed by a positive integer, which creates a *shallow clone*. A shallow clone is one where only the specified number of revisions are downloaded from the remote repository; it's limited, because it can't be cloned/fetched/pushed from or pushed to. This can be useful for reducing the clone time for very large repositories.

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

Figure 2.5 shows the state of the repository after `git clone`. It's identical to the state after `git push` in figure 2.4. This shows that the clone was successful and the newly created local repository has the same contents as the deleted old local repository.

6b437c77	🔗 master	origin/master	Add opening joke. Funny?	Mike McQuaid	29 September 2013 11:30
6576b680	🔗		Initial commit of book.	Mike McQuaid	29 September 2013 10:30

Figure 2.5 Local repository after `git clone`

Cloning a repository has also created a new remote called `origin`. `origin` is the default remote and references the repository from which the clone originated (which is <https://github.com/GitInPractice/GitInPracticeRedux.git> in this case).

Now let's learn how to pull new commits from the remote repository.

Technique 9 Pulling changes from another repository: `git pull`

`git pull` downloads the new commits from another repository and merges the remote branch into the current branch. If you run `git pull` on the local repository, you just see a message stating `Already up-to-date`. `git pull` in this case contacted the remote repository, saw that there were no changes to be downloaded, and let you know that it was up to date. This is expected, because this repository has been pushed to but not updated since.

To test `git pull`, let's create another clone of the same repository, make a new commit, and `git push` it. This will allow downloading new changes with `git pull` on the original remote repository.

To create another cloned, local repository and push a commit from it, do the following:

- 1 Change to the directory where you want the new `GitInPracticeRedux` repository to be created—for example, `cd /Users/mike/` to create the new local repository in `/Users/mike/GitInPracticeReduxPushTest`.
- 2 Run `git clone` with your repository URL and destination directory appended. So if your username is `GitInPractice`, your repository is `GitInPracticeRedux`, and the destination directory is named `GitInPracticeReduxPushTest`, run `git clone https://github.com/GitInPractice/GitInPractice-Redux.git GitInPracticeReduxPushTest` to clone into the `GitInPracticeReduxPushTest` directory.
- 3 Change directory to the new Git repository: for example, `cd /Users/mike/GitInPracticeReduxPushTest/`.
- 4 Modify the `GitInPractice.asciidoc` file.
- 5 Run `git add GitInPractice.asciidoc`.
- 6 Run `git commit --message 'Improve joke comic timing.'`
- 7 Run `git push`.

Now that you've pushed a commit to the `GitInPracticeRedux` remote on GitHub, you can change back to your original repository and `git pull` from it. Keep the `GitInPracticeReduxPushTest` directory, because you'll use it later.

Problem

You wish to pull new commits into the current branch on the local `GitInPractice-Redux` repository from the remote repository on GitHub.

Solution

- 1 Change directory to the original Git repository: for example, `cd /Users/mike/GitInPracticeRedux/`.
- 2 Run `git pull`. The output should resemble the following.

Listing 2.5 Output: pulling new changes

```
# git pull
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 3 (delta 0)
Unpacking objects: 100% (3/3), done.
From https://github.com/GitInPractice/GitInPracticeRedux
 6b437c7..85a5db1  master    -> origin/master
Updating 6b437c7..85a5db1
Fast-forward
 GitInPractice.asciidoc | 5 +++--
 1 file changed, 3 insertions(+), 2 deletions(-)
```

1 Object preparation and transmission

2 Remote URL

3 Remote branch update

4 Local branch update

5 Merge type

6 Lines changed in the file

7 Diff summary

- ❶ can be safely ignored again.
- ❷ matches the remote repository URL you saw used for `git push`.
- ❸ shows how the state of the `origin` remote's `master` branch was updated, and that this can be seen in `origin/master`. `origin/master` is a valid ref that can be used with tools such as `git diff`, so `git diff origin/master` shows the differences between the current working tree state and the `origin` remote's `master` branch.
- ❹ shows that after `git pull` downloaded the changes from the other repository, it merged the changes from the tracking branch into the current branch. In this case, your `master` branch had the changes from the `master` branch on the remote `origin` merged in. You can see in this case that the SHA-1s match those in ❸. It has been updated to include the new commit (85a5db1).
- ❺ was a *fast-forward merge*, which means no merge commit was made. Fast-forward merges will be explained in technique 14.
- ❻ is the same as the lines changed from `git commit` in technique 3 or `git diff` in technique 5. It shows a summary of the changes that have been pulled into your `master` branch.
- ❼ is the same as the diff summary from `git commit` in technique 3 or `git diff` in technique 5.

Discussion

`git pull` can take a `--rebase` flag that performs a rebase rather than a merge. This will be covered later in technique 45.

WHY DID A MERGE HAPPEN? It may be confusing that a merge has happened here. Didn't you just ask for the updates from that branch? You haven't created any other branches, so why did a merge happen? In Git, all remote branches (including the default `master` branch) are linked to your local branches only if the local branch is tracking the remote branch. As a result, when you're pulling changes from a remote branch into your current branch, the result may sometimes be a situation where you've made local changes and the remote branch has also received changes. In this case, a merge must be made to reconcile the differing local and remote branches.

You can see from figure 2.6 that a new commit has been added to the repository and that both `master` and `origin/master` have been updated.

You've pulled the new commits from the `GitInPracticeRedux` remote repository into your local repository, and Git has merged them into your `master` branch. Now let's learn how to download changes without applying them onto your `master` branch.

85a5db18	🔗	master	origin/master	Improve joke comic timing.	Mike McQuaid	30 September 2013 17:30
6b437c77	🔗			Add opening joke. Funny?	Mike McQuaid	29 September 2013 11:30
6576b680	🔗			Initial commit of book.	Mike McQuaid	29 September 2013 10:30

Figure 2.6 Local repository after `git pull`

Technique 10 Fetching changes from a remote without modifying local branches: *git fetch*

Remember that `git pull` performs two actions: fetching the changes from a remote repository and merging them into the current branch. Sometimes you may wish to download the new commits from the remote repository without merging them into your current branch (or without merging them yet). To do this, you can use the `git fetch` command. `git fetch` performs the fetching action of downloading the new commits but skips the merge step (which you can manually perform later).

To test `git fetch`, let's use the `GitInPracticeReduxPushTest` local repository again to make another new commit and `git push` it. This will allow downloading new changes with `git fetch` on the original remote repository.

To push another commit from the `GitInPracticeReduxPushTest` repository, do the following:

- 1 Change directory to the `GitInPracticeReduxPushTest` repository; for example, `cd /Users/mike/GitInPracticeReduxPushTest/`.
- 2 Modify the `GitInPractice.asciidoc` file.
- 3 Run `git add GitInPractice.asciidoc`.
- 4 Run `git commit --message 'Joke rejected by editor!'`.
- 5 Run `git push`.

Now that you've pushed another commit to the `GitInPracticeReduxPushTest` remote on GitHub, you can change back to your original repository and `git fetch` from it. If you wish, you can now delete the `GitInPracticeReduxPushTest` repository by running a command like `rm -rf /Users/mike/GitInPracticeReduxPushTest/`.

Problem

You wish to fetch new commits to the local `GitInPracticeRedux` repository from the `GitInPracticeRedux` remote repository on GitHub without merging into your master branch.

Solution

- 1 Change directory to the Git repository: `cd /Users/mike/GitInPracticeRedux/`.
- 2 Run `git fetch`. The output should resemble the following.

Listing 2.6 Output: fetching new changes

```
# git fetch
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 3 (delta 0)
Unpacking objects: 100% (3/3), done.
From https://github.com/GitInPractice/GitInPracticeRedux
  85a5db1..07fc4c3  master    -> origin/master
```

Object preparation and transmission

Remote URL

Remote branch update

The `git fetch` output is the same as the first part of the `git pull` output. But the SHA-1s are different again, because a new commit was downloaded. This is because `git fetch` does effectively half of what `git pull` is doing. If your `master` branch is tracking the `master` branch on the remote origin, then `git pull` is directly equivalent to running `git fetch && git merge origin/master`.

You’ve fetched the new commits from the remote repository into your local repository without merging them into your `master` branch.

Discussion

You can see from figure 2.7 that another new commit has been added to the repository, but this time only `origin/master` has been updated; `master` has not. To see this, you may need to select the `origin` remote and `master` remote branch in the GitX sidebar. Selecting commits by remote branches is a feature sadly not available in `gitk`.

BRANCHES	Short SHA	Subject	Author	Date
master	07fc4c3c	Joke rejected by editor!	Mike McQuaid	11 October 2013 18:30
	85a5db18	Improve joke comic timing.	Mike McQuaid	30 September 2013 17:30
	6b437c77	Add opening joke. Funny?	Mike McQuaid	29 September 2013 11:30
	6576b680	Initial commit of book.	Mike McQuaid	29 September 2013 10:30

Figure 2.7 Remote repository after `git fetch`

To clean up the local repository, let’s do another quick `git pull` to update the state of the `master` branch based on the (already fetched) `origin/master`. To pull new commits into the current branch on the local `GitInPracticeRedux` repository from the remote repository on GitHub, do the following:

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

Listing 2.7 Output: pull after fetch

```
# git pull
Updating 85a5db1..07fc4c3  ← Local branch update
Fast-forward             ← Merge type
  GitInPractice.asciidoc | 4 +---  ← Lines changed in the file
  1 file changed, 1 insertion(+), 3 deletions(-)  ← Diff summary
```

This shows the latter part of the first `git pull` output you saw. No more changes were fetched from the `origin` remote, and the local `master` branch hadn’t been updated. As a result, this `git pull` behaved the same as running `git merge origin/master`.

Figure 2.8 shows that the `master` branch has been updated to match the `origin/master` latest commit once more.

07fc4c3c	● master	origin/master	Joke rejected by editor!	Mike McQuaid	11 October 2013 18:30
85a5db18	●		Improve joke comic timing.	Mike McQuaid	30 September 2013 17:30
6b437c77	●		Add opening joke. Funny?	Mike McQuaid	29 September 2013 11:30
6576b680	●		Initial commit of book.	Mike McQuaid	29 September 2013 10:30

Figure 2.8 Local repository after git fetch and then git pull

SHOULD YOU USE PULL OR FETCH? I prefer to use git fetch over git pull. This means I can continue to fetch regularly in the background and only include these changes in my local branches when it's convenient and using the method I find most appropriate, which may be merging or rebasing (or resetting, which you will see later in technique 42). Additionally, I sometimes work in situations where I have no internet connection (such as on planes), and using git fetch is superior in these cases; it can fetch changes without requiring any human interaction in the case of a merge conflict, for example.

We've talked about local branches and remote branches but haven't created any. Let's learn about how branches work and how to create them.

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

When committing in Git, the history continues linearly; what was the most recent commit becomes the parent commit for the new commit. This parenting continues back to the initial commit in the repository. You can see an example of this in figure 2.9.

Sometimes this linear approach isn't enough for software projects. Sometimes you may need to make new commits that aren't yet ready for public consumption. This requires *branches*.

Branching allows two independent tracks through history to be created and committed to without either modifying the other. You can happily commit to an independent branch without the fear of disrupting the work of another branch. This means you can, for example, commit broken or incomplete features rather than having to

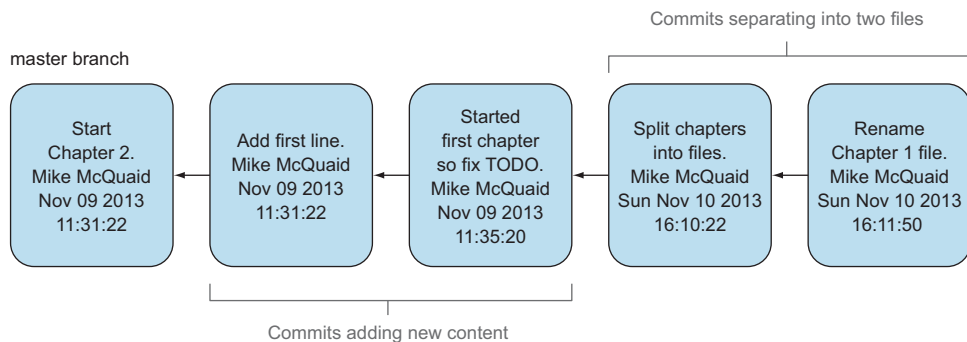


Figure 2.9 Committing without using branches

wait for others to be ready for their commits. It also means your changes can be isolated from changes made by others until you're ready to integrate them into your branch. Figure 2.10 shows the same commits as figure 2.9 if they were split between two branches instead for isolation.

When a branch is created and new commits are made, that branch advances to include the new commits. In Git, a branch is no more than a pointer to a particular commit. This is unlike other version control systems such as Subversion, in which branches are subdirectories of the repository.

The branch is pointed to a new commit when a new commit is made on that branch. A *tag* is similar to a branch, but it points to a single commit and remains pointing to the same commit even when new commits are made. Typically tags are used for annotating commits; for example, when you release version 1.0 of your software, you may tag the commit used to build the 1.0 release with a 1.0 tag. This means you can come back to it in the future, rebuild that release, or check how certain things worked without fear that it will be somehow changed automatically.

Branching allows two independent tracks of development to occur at once. In Figure 2.10, the `separate-files` branch was used to separate the content from a single file and split it into two new files. This allowed refactoring of the book structure to be done in the `separate-files` branch while the default branch (known as `master` in Git) could be used to create more content. In version control systems like Git, where creating a branch is a quick, local operation, branches may be used for every independent change.

Some programmers create new branches whenever they work on a new bug fix or feature and then integrate these branches at a later point, perhaps after requesting that others review their changes. This means even for programmers working without a team, it can be useful to have multiple branches in use at any one point. For example,

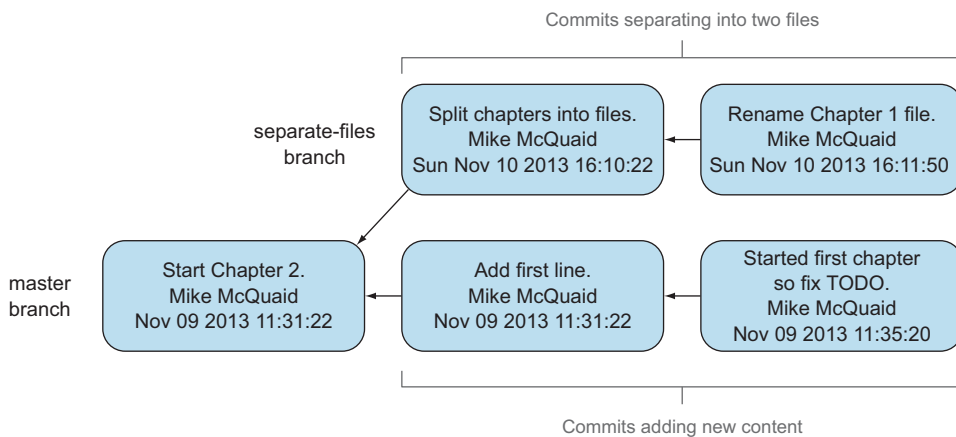


Figure 2.10 Committing to multiple branches

you may be working on a new feature but realize that a critical error in your application needs to be fixed immediately. You can quickly create a new branch based off the version used by customers, fix the error, and switch back to the branch to which you'd been committing the new feature.

Problem

You wish to create a new local branch named `chapter-two` from the current (`master`) branch.

Solution

- 1 Change directory to the Git repository: for example, `cd /Users/mike/GitInPracticeRedux/`.
- 2 Run `git branch chapter-two`. There will be no output.

You can verify that the branch was created by running `git branch`, which should have the following output.

Listing 2.8 Output: listing branches

```
# git branch
chapter-two      ← 1 New branch
* master         ← 2 Current branch
```

- 1 was created with the expected name.
- 2 is indicated by the `*` prefix, which shows you're still on the `master` branch as before. `git branch` creates a new branch but doesn't change to it.

You've created a new local branch named `chapter-two` that currently points to the same commit as `master`.

Discussion

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

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

You can see from figure 2.11 that there's a new branch label for the `chapter-two` branch. In the GitX GUI, the label colors indicate the following:

- *Orange*—The currently checked-out local branch
- *Green*—A non-checked-out local branch
- *Blue*—A remote branch

07fc4c3c	chapter-two	master	origin/master	Joke rejected by editor!	Mike McQuaid	11 October 2013 18:30
85a5db18				Improve joke comic timing.	Mike McQuaid	30 September 2013 17:30
6b437c77				Add opening joke. Funny?	Mike McQuaid	29 September 2013 11:30
6576b680				Initial commit of book.	Mike McQuaid	29 September 2013 10:30

Figure 2.11 Local repository after `git branch chapter-two`

Note that print editions of this book are grayscale, so these colors are not visible. Instead, please compare them to GitX on your computer. Figure 2.12 shows how these two branch pointers point to the same commit.

You’ve seen how `git branch` creates a local branch but doesn’t change to it. To do that requires using `git checkout`.

CAN BRANCHES BE NAMED ANYTHING? A branch can’t have spaces or two consecutive dots (..) anywhere in its name, so `chapter..two` is an invalid branch name and `git branch` will refuse to create it. The dots case is due to the special meaning of .. for a commit range for the `git diff` command (which you saw used in section 1.7).

WHAT NAMES SHOULD YOU USE FOR BRANCHES? Name branches according to their contents. For example, the `chapter-two` branch you created indicates that the commits in this branch will reference the second chapter. I recommend a format of describing the branch’s purpose in multiple words separated by hyphens. For example, a branch that is performing cleanup on the test suite should be named something like `test-suite-cleanup`.

Technique 12 Checking out a local branch: *git checkout*

Once you’ve created a local branch, you’ll want to check out the contents of another branch into Git’s working directory. The state of all the current files in the working directory will be replaced with the new state based on the revision to which the new branch currently points.

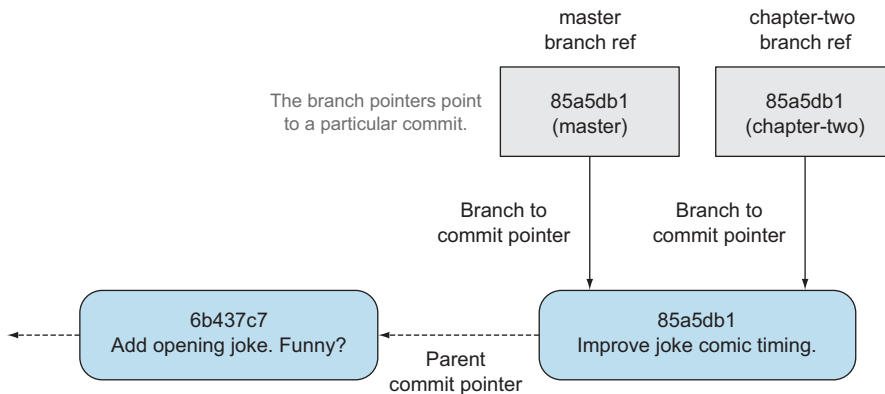


Figure 2.12 Branch pointers

Problem

You wish to change to a local branch named `chapter-two` from the current (`master`) branch.

Solution

- 1 Change directory to the Git repository; for example, `cd /Users/mike/GitInPracticeRedux/`.
- 2 Run `git checkout chapter-two`. The output should be `Switched to branch 'chapter-two'`.

You've checked out the local branch named `chapter-two` and moved from the `master` branch.

Discussion

As mentioned earlier, some other version control systems (such as Subversion) use `checkout` to refer to the initial download from a remote repository, but `git checkout` is used here to change branches. This may be slightly confusing until we look at Git's full remote workflow. Figure 2.13 shows Git's local workflow again. Under closer examination, `git checkout` and `svn checkout` behave similarly; both check out the contents of a version control repository into the working directory, but Subversion's repository is remote and Git's repository is local. In this case, `git checkout` is requesting the checkout of a particular branch so the current state of that branch is checked out into the working directory.

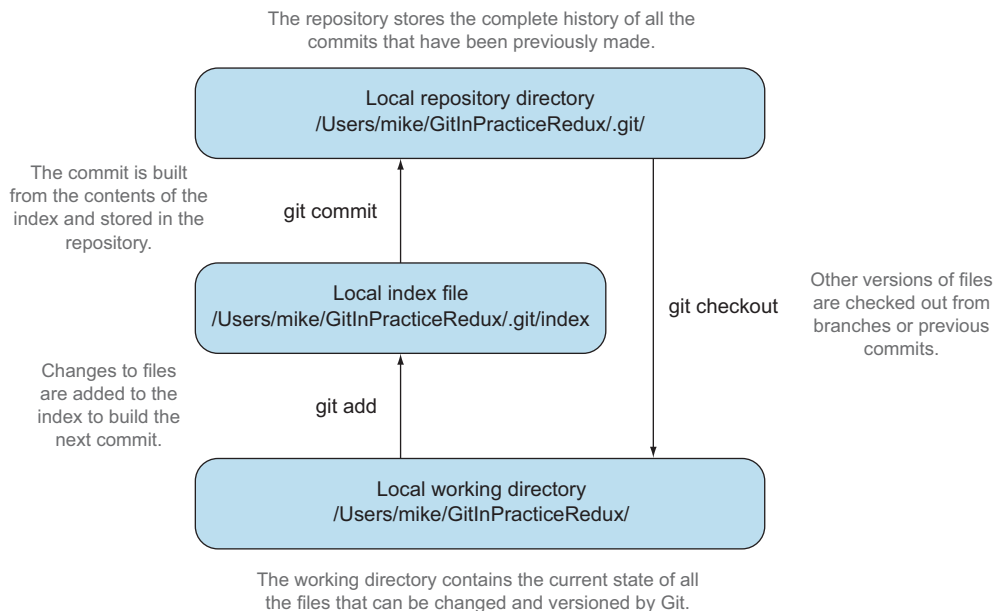


Figure 2.13 Git `add/commit/checkout` workflow

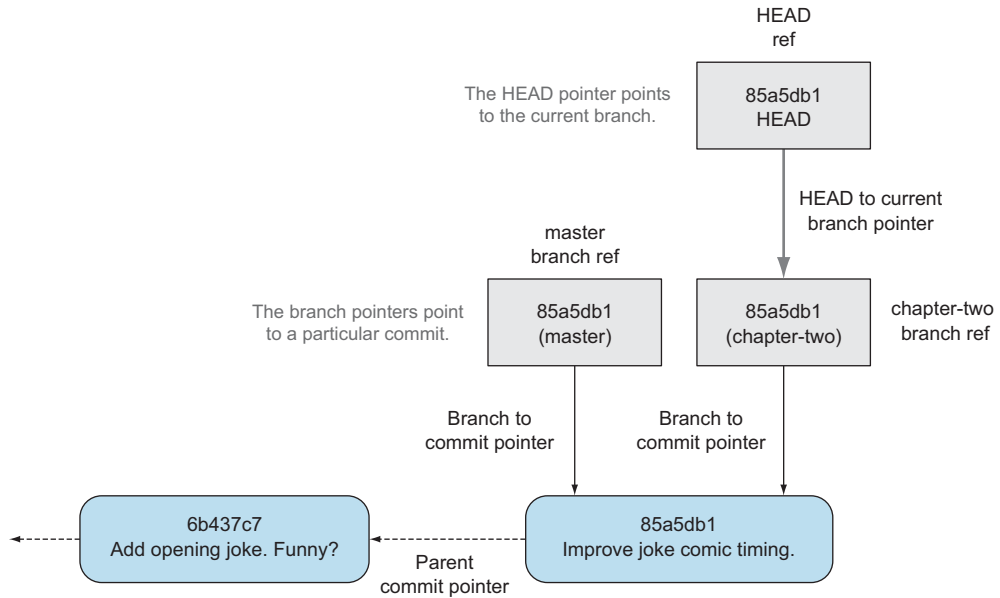


Figure 2.14 HEAD pointer with multiple branches

Afterward, the HEAD pointer (shown in figure 2.14) is updated to point to the current, chapter-two branch pointer, which in turn points to the top commit of that branch. The HEAD pointer moved from the master to the chapter-two branch when you ran `git checkout chapter-two`, setting chapter-two to be the current branch.

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

Technique 13 *Pushing a local branch remotely*

Now that you've created a new branch and checked it out, it would be useful to push any new commits made to the remote repository. To do this requires using `git push` again.

Problem

You wish to push the changes from the local chapter-two branch to create the remote branch chapter-two on GitHub.

Solution

- 1 Change directory to the Git repository, such as `cd /Users/mike/GitInPracticeRedux/`.
- 2 Run `git checkout chapter-two` to ensure that you're on the chapter-two branch.
- 3 Run `git push --set-upstream origin chapter-two`. The output should resemble the following.

Listing 2.9 Output: pushing and setting an upstream branch

```
git push --set-upstream origin chapter-two
Total 0 (delta 0), reused 0 (delta 0)
To https://github.com/GitInPractice/GitInPracticeRedux.git
 * [new branch]      chapter-two -> chapter-two
Branch chapter-two set up to track remote branch
chapter-two from origin.
```

1 Object preparation and transmission

2 Local/remote branch

3 Set tracking branch

The output is much the same as it was for the previous `git push` run:

- 1 (although still ignorable) shows that no new objects were sent. The reason is that the chapter-two branch still points to the same commit as the master branch; it's effectively a different name (or, more accurately, ref) pointing to the same commit. As a result, no more commit objects have been created, and therefore no more were sent.
- 2 has chapter-two as the branch name.
- 3 has chapter-two as the branch name.

You've pushed your local chapter-two branch and created a new remote branch named chapter-two on the remote repository.

Discussion

Remember that now the local chapter-two branch is tracking the remote chapter-two branch, so any future `git pull` or `git push` on the chapter-two branch will use the origin remote's chapter-two branch. As you'll hopefully have anticipated, figure 2.15 shows the addition of another remote branch named origin/chapter-two.

07fc4c3c	chapter-two	master	origin/chapter-two	origin/master	Joke rejected by editor!	Mike McQuaid	11 October 2013 18:30
85a5db18					Improve joke comic timing.	Mike McQuaid	30 September 2013 17:30
6b437c77					Add opening joke. Funny?	Mike McQuaid	29 September 2013 11:30
6576b680					Initial commit of book.	Mike McQuaid	29 September 2013 10:30

Figure 2.15 Local repository after `git push --set-upstream origin chapter-two`

Technique 14 Merging an existing branch into the current branch: *git merge*

At some point, you have a branch you're done with, and you want to bring all the commits made on it into another branch. This process is known as a *merge*.

When a merge is requested, all the commits from another branch are pulled into the current branch. Those commits then become part of the history of the branch. Note from figure 2.16 that the commit in which the merge is made has two parent commits rather than one; it joins two separate paths through the history into a single one. After a merge, you may decide to keep the existing branch to add more commits to it and perhaps merge again at a later point (only the new commits will need to be merged next time). Alternatively, you can delete the branch and make future commits on Git's default master branch, and create another branch when needed in the future.

Problem

You wish to make a commit on the local branch named `chapter-two` and merge this into the master branch.

Solution

- 1 Change directory to the Git repository; for example, `cd /Users/mike/GitInPracticeRedux/`.
- 2 Run `git checkout chapter-two` to ensure that you're on the `chapter-two` branch.
- 3 Modify the contents of `GitInPractice.asciidoc`, and run `git add GitInPractice.asciidoc`.
- 4 Run `git commit --message 'Start Chapter 2.'`

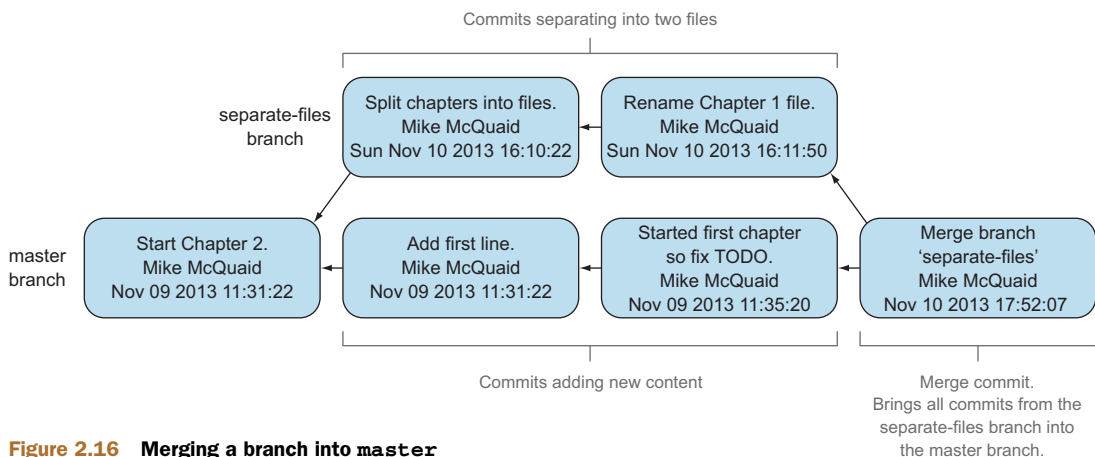


Figure 2.16 Merging a branch into master

- 5 Run `git checkout master` to check out the branch you wish to merge chapter-two into.
- 6 Run `git merge chapter-two`. The output should resemble the following.

Listing 2.10 Output: merging a branch

```
# git merge chapter-two
Merge type 2 Updating 07fc4c3..ac14a50      ← 1 Local branch update
Fast-forward
GitInPractice.asciidoc | 2 ++
1 file changed, 2 insertions(+)      ← 3 Diff summary
```

The output may seem familiar from the `git pull` output. Remember, this is because `git pull` actually does a `git fetch` & `git merge`:

- 1 shows the changes that have been merged into the local `master` branch. Note that the SHA-1 has been updated from the previous `master` SHA-1 (07fc4c3) to the current `chapter-two` SHA-1 (ac14a50).
- 2 was a *fast-forward merge*. This means no merge commit (a commit with multiple parents) was needed, so none was made. The `chapter-two` commits were made on top of the `master` branch, but no more commits had been added to the `master` branch before the merge was made. In Git's typical language: the merged commit (tip of the `chapter-two` branch) is a descendant of the current commit (tip of the `master` branch). If there had been another commit on the `master` branch before merging, then this merge would have created a merge commit. If there had been conflicts between the changes made in both branches that couldn't automatically be resolved, then a merge conflict would have been created and had to be resolved.
- 3 shows a summary of the changes that have been merged into your `master` branch from the `chapter-two` branch.

You've merged the `chapter-two` branch into the `master` branch.

Discussion

This brings the commit that was made in the `chapter-two` branch into the `master` branch.

WHAT IF YOU TRY TO MERGE THE SAME COMMIT INTO A BRANCH MULTIPLE TIMES?

`git merge` won't merge the same commit into a branch multiple times; it will exit and output `Already up-to-date.` rather than performing the merge.

You can see from figure 2.17 that the `chapter-two` and `master` branches point to the same commit once more.

ac14a504	chapter-two	master	Start Chapter 2.	Mike McQuaid	9 November 2013 11:31
07fc4c3c	origin/chapter-two	origin/master	Joke rejected by editor!	Mike McQuaid	11 October 2013 18:30
85a5db18			Improve joke comic timing.	Mike McQuaid	30 September 2013 17:30
6b437c77			Add opening joke. Funny?	Mike McQuaid	29 September 2013 11:30
6576b680			Initial commit of book.	Mike McQuaid	29 September 2013 10:30

Figure 2.17 Local repository after `git merge chapter-two`

2.2 Merge conflicts

So far, merges may have sounded too good to be true; you can work on multiple things in parallel and combine them at any later point in any order. Not so fast, my merge-happy friend; I haven't told you about merge conflicts yet.

A *merge conflict* occurs when both branches involved in the merge have changed the same part of the same file. Git will try to automatically resolve these conflicts but sometimes is unable to do so without human intervention. This case produces a merge conflict.

Listing 2.11 Merge conflict in Git

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

## Chapter 2
// TODO: write two chapters                  ← ❹ Branch separator
=====
// TODO: think of funny first line that editor will approve. ← ❺ Current
>>>>>> separate-files                       ← ❻ Current marker
```

When a merge conflict occurs, the version control system goes through any files that have conflicts and inserts something similar to the preceding markers. These markers indicate the versions of the file on each branch:

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

HOW CAN CONFLICT MARKERS BE FOUND QUICKLY? When searching a large file for the merge-conflict markers, you can enter <<<< in your text editor's Find tool to locate them quickly.

The person performing the merge must manually edit the file to produce the correctly merged output, save it, and mark the merge as resolved. Sometimes, resolving the conflict involves picking all the lines of a single version: either the previous version's lines or the new branch's lines. Other times, resolving the conflict involves combining some lines from the previous version and some lines from the new branch. In cases where other files have been edited (as in this example), it may also involve putting some of these lines into other files.

When conflicts have been resolved, a *merge commit* can be made. This stores the two parent commits and the conflicts that were resolved so they can be inspected in the future. Unfortunately, sometimes people pick the wrong option or merge incorrectly, so it's good to be able to later see what conflicts they had to resolve.

We'll cover resolving merge conflicts in more detail later in technique 34.

2.3 Rebasing

A *rebase* is a method of rewriting history in Git that is similar to a merge. A rebase involves changing the parent of a commit to point to another.

Figure 2.18 shows a rebase of the `separate-files` branch onto the `master` branch. The rebase operation has changed the parent of the first commit in the `separate-files` branch to be the last commit in the `master` branch. This means all the content changes from the `master` branch are now included in the `separate-files` branch, and any conflicts were manually resolved but weren't stored (as they would be in a merge conflict).

We'll cover rebasing in more detail later in technique 43. All you need to remember for now is that it's a different approach to a merge that can be used for a similar outcome (pulling changes from one branch into another).

Technique 15 Deleting a remote branch

Now that the `chapter-two` branch has been merged into the `master` branch, the new commit that was made in the `chapter-two` branch is in the `master` branch. This means you can push the `master` branch to push all the `chapter-two` changes to `origin/master`. Once this is done (and assuming you don't want to make any more commits to the `chapter-two` branch), `origin/chapter-two` can be safely deleted.

WHY DELETE THE BRANCHES? Sometimes branches in version control systems are kept for a long time, and sometimes they're temporary. A long-running branch may be one that represents the version deployed to a particular server. A short-running branch may be a single bug fix or feature that has been completed. In Git, once a branch has been merged, the history of the branch is still visible in the history, and the branch can be safely deleted,

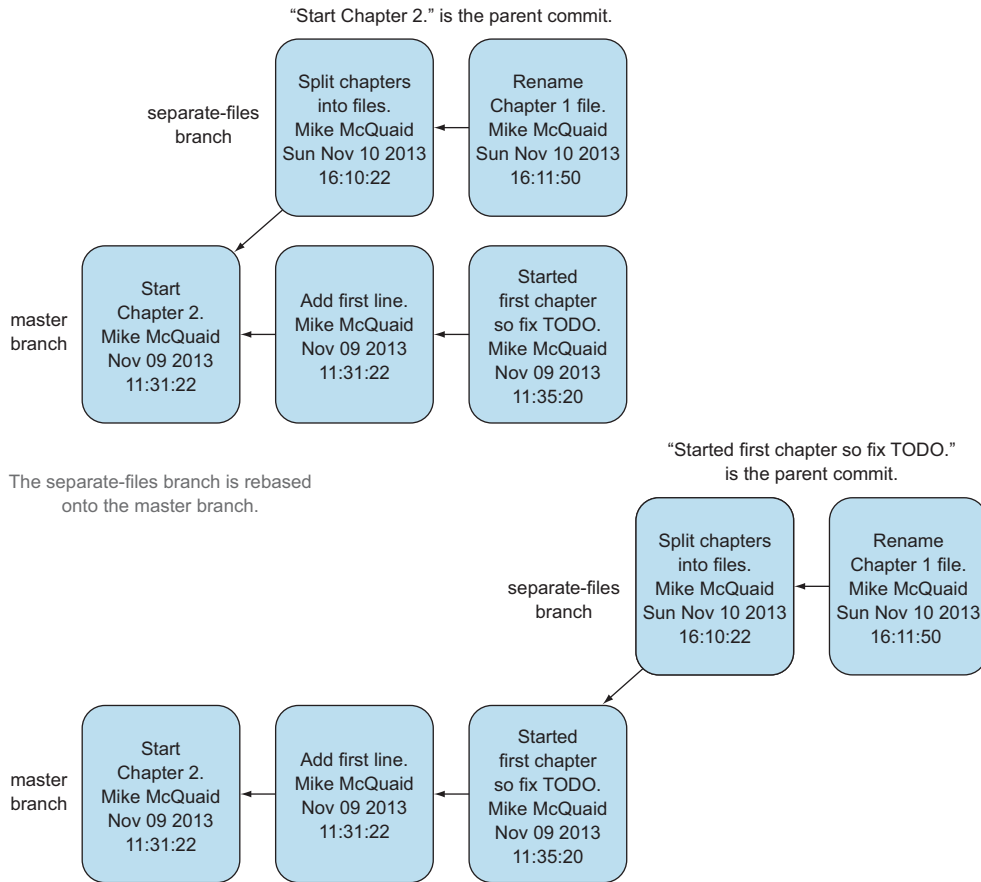


Figure 2.18 Rebasing a branch on top of master

because a merged branch is, at that point, just a ref to an existing commit in the history of the branch it was merged into.

Problem

You wish to push the current master branch and delete the branch named `chapter-two` on the remote origin.

Solution

- 1 Change directory to the Git repository; for example, `cd /Users/mike/GitInPracticeRedux/`.
- 2 Run `git checkout master` to ensure that you're on the master branch.
- 3 Run `git push`.
- 4 Run `git push --delete origin chapter-two`. The output should resemble the following.

Listing 2.12 Output: deleting a remote branch

```
# git push origin :chapter-two

To https://github.com/GitInPractice/GitInPracticeRedux.git <1 Remote URL
- [deleted] chapter-two <2 Deleted branch
```

1 shows the remote repository from which the branch was deleted.

2 shows the name of the branch (chapter-two) that has been deleted from the remote repository.

You have deleted the chapter-two branch from the remote repository.

Discussion

In figure 2.19, you can see that origin/master has been updated to the same commit as master and that origin/chapter-two has been removed.

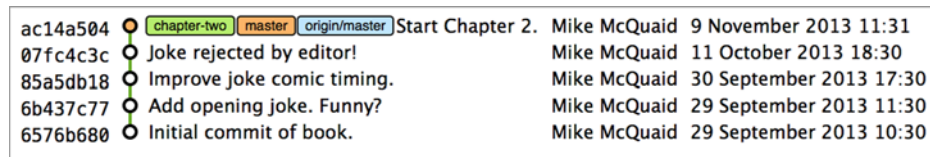


Figure 2.19 Local repository after `git push origin :chapter-two`

Technique 16 Deleting the current local branch after merging

The chapter-two branch has all its commits merged into the master branch, and the remote branch has been deleted, so the local branch can now be deleted too.

Problem

You wish to delete the local branch named chapter-two.

Solution

- 1 Change directory to the Git repository; for example, `cd /Users/mike/GitInPracticeRedux/`.
- 2 Run `git checkout master` to ensure that you're on the master branch.
- 3 Run `git branch --delete chapter-two`. The output should be `Deleted branch chapter-two (was ac14a50)`.

You've deleted the chapter-two branch from the local repository.

Discussion

Figure 2.20 shows the final state with all evidence of the chapter-two branch removed (other than the commit message).

ac14a504	●	master	origin/master	Start Chapter 2.	Mike McQuaid	9 November 2013 11:31
07fc4c3c	○			Joke rejected by editor!	Mike McQuaid	11 October 2013 18:30
85a5db18	○			Improve joke comic timing.	Mike McQuaid	30 September 2013 17:30
6b437c77	○			Add opening joke. Funny?	Mike McQuaid	29 September 2013 11:30
6576b680	○			Initial commit of book.	Mike McQuaid	29 September 2013 10:30

Figure 2.20 Local repository after `git branch --delete chapter-two`

WHY DELETE THE REMOTE BRANCH BEFORE THE LOCAL BRANCH? You had merged all the chapter-two changes into the master branch and pushed this to origin/master. As a result, the chapter-two and origin/chapter-two branches are no longer needed. But Git will refuse to delete a local branch with `git branch --delete` if it hasn't been merged into the current branch or its changes haven't been pushed to its tracking branch (origin/chapter-two in this case). Deleting origin/chapter-two first means the local chapter-two branch can be deleted with `git branch --delete` without Git complaining that chapter-two has changes that need to be pushed to origin/chapter-two.

2.4 Summary

In this chapter, you learned the following:

- How to push your local repository to a remote repository
- How to clone an existing remote repository
- How to push and pull changes to/from a remote repository
- That fetching allows you to obtain changes without modifying local branches
- That pulling is equivalent to fetching and then merging
- How to check out local and remote branches
- How to merge branches and then delete from the local and remote repositories

Part 2

Git essentials

Part 2 (chapters 3–6) will cover the most essential commands to learn when using Git either alone or with a team. It's here that you'll start to see some of the more powerful differences between Git and other version control systems, including advanced branching capabilities and history rewriting.

This part covers the following topics:

- How to delete, move, and rename versioned files and directories
- How to ignore, reset, and temporarily stash certain files and changes
- How to tweak history output to display only the information you need
- How to find out what person or commit caused a particular bug
- How to easily resolve merge conflicts
- How to tag new versions of software
- How to cherry-pick or revert individual commits
- How to view Git's history even after rewriting
- How to re-create branches and commits on top of other commits
- How to avoid losing your work

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 ← rename GitInPractice.asciidoc => 01-IntroducingGitInPractice.asciidoc (100%)

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
```



Commit message

1 line deleted

Deleted filename

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

History visualization



This chapter covers

- Filtering `git log` output commits
- Formatting `git log` output to display the information you care about
- Finding why and when a line in a file was changed, and by whom, using `git blame`
- Identifying which commit caused a particular bug using `git bisect`

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

Let's start by learning how to optimize your use of `git log` to list only particular commits.

Technique 29 Listing only certain commits

Sometimes, when examining history, you'll want to filter the commits that are displayed based on some of their metadata. Perhaps you're tracking down a commit that you can remember was made on a rough date, by a particular person, or with a particular word in its commit message. You could do this manually, but sometimes there are too many commits in the history to scan through the `git log` or `gitx` output in a timely fashion. For these cases, the `git log` command has various flags and arguments that you can use to filter which commits are shown in its output.

Let's start by trying to find a commit by author, date, and commit message simultaneously.

Problem

You want to list the commits authored by Mike McQuaid after November 10, 2013, with the string "file." in their message.

Solution

- 1 Change to the directory containing your repository; for example,
`cd /Users/mike/GitInPracticeRedux/.`
- 2 Run `git log --author "Mike McQuaid" --after "Nov 10 2013" --grep 'file\.'` and, if necessary, `q` to exit. The output should resemble the following.

Listing 4.1 Output: filtered log

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

commit 06b5eb58b62ba8bbbee258705c636ca7ac20b49
Author: Mike McQuaid <mike@mikemcquaid.com>
Date: Thu Nov 28 15:49:30 2013 +0000

    Remove unfavourable review file.

commit fcd8f6e957a03061cdf411851fe38034a44c97ab
Author: Mike McQuaid <mike@mikemcquaid.com>
Date: Thu Nov 28 15:48:24 2013 +0000

    Add first review temporary file.

commit c6eed6681efc8d0bff908e6dbb7d887c4b3fab3e
Author: Mike McQuaid <mike@mikemcquaid.com>
Date: Thu Nov 28 15:39:38 2013 +0000

    Rename book file to first part file.
```

← Most recent
commit

← Least recent
commit

The filtered log output is essentially the same as the `git log` output you saw in technique 4, but with only the commits that are matched by the specified arguments.

The arguments provided to the `log` command indicate the following:

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

McQuaid <mike@mikemcquaid.com> and found a match because the string started with the requested *Mike McQuaid*.

- `--after` (or `--since`) specifies that the only commits shown should be those that were made after the specified date. These dates can be in any format that Git recognizes, such as Nov 10 2013, 2014-01-30, midnight, or yesterday.
- `--grep` specifies a regular expression that matches the contents of the commit message. `file\.` was used rather than `file.` to escape the `.` character.

WHAT ARE REGULAR EXPRESSIONS? Regular expressions are search patterns that are typically used to match patterns inside strings. There are multiple variants of regular expressions, and Git uses the POSIX regular expression format. Git uses them for various filtering operations such as filtering by commit message or author, as you saw previously. Characters can have literal meanings, such as `a`, which will always match a lowercase `a` character; or they can have special meanings, such as `.`, which will match any character. For example, `file.` will match the string *files* or *filed*. To turn a character with a special meaning into one with a literal meaning, you can escape it with a backslash: `file\.` will match the string *file.* but not the string *files*. I'm not going to cover regular expressions in detail because they're beyond the scope of this book. Don't worry if you don't fully understand them; what you've seen here should be more than enough to work with Git. If you wish to learn more about them, I recommend the Wikipedia page http://en.wikipedia.org/wiki/Regular_expression.

You've shown a subset of commits filtered by author, date, and commit message.

Discussion

`git log` can take the following arguments:

- A `--max-count` (or `-n`) argument to limit the number of commits shown in the log output. I tend to use this often when I only care about something in, say, the last 10 commits and don't want to scroll through more output than that.
- A `--reverse` argument to show the commits in ascending chronological order (oldest commit first).
- A `--before` (or `--until`) argument, which will only show commits before the given date. This is the reverse of `--after`.
- A `--merges` flag (or `--min-parents=2`), which will only show *merge commits*—commits that have at least two parents. If you adopt a branch-heavy workflow with Git, this will be useful in identifying which branches were merged and when.

4.1 *git show*

`git show` is a command similar to `git log`, but it shows a single commit. It also defaults to showing what was changed in that commit. Remember from technique 4 that `git log` has a `--patch` (or `-p`) argument to show what was changed by each commit in its output.

Listing 4.2 Output: showing a commit

```
# git show HEAD^

commit 06b5eb58b62ba8bbbeee258705c636ca7ac20b49
Author: Mike McQuaid <mike@mikemcquaid.com>
Date: Thu Nov 28 15:49:30 2013 +0000

    Remove unfavourable review file.

diff --git a/GitInPracticeReviews.tmp b/GitInPracticeReviews.tmp
deleted file mode 100644
index ebb69c3..0000000
--- a/GitInPracticeReviews.tmp
+++ /dev/null
@@ -1,0,0 @@
-Git Sandwich
```

← 1 **Commit information**

← 2 **Commit diff**

① shows all the same information expected in `git log` output, but it only ever shows a single commit.

② shows the changes made in that commit. It's the equivalent of typing `git diff HEAD^..HEAD^`—the difference between the previous commit and the one before it.

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

Technique 30 Listing commits with different formatting

The default `git log` output format is helpful, but it takes a minimum of six lines of output to display each commit. It displays the commit SHA-1, author name and email, commit date, and the full commit message (each additional line of which adds a line to the `git log` output). Sometimes you'll want to display more information, and sometimes you'll want to display less. You may even have a personal preference about how the output is presented that doesn't match how it currently is.

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

WHY ARE COMMITS STRUCTURED LIKE EMAILS? Remember in technique 4 I mentioned that commits are structured like emails? This is because Git was initially created for use by the Linux kernel project, which has a high-traffic mailing list. People frequently send commits (known as *patches*) to the mailing list. Previously there was an implicit format that people used to turn a requested change into an email for the mailing list, but Git can convert commits to and from an email format to facilitate this. Commands such as `git format-patch`, `git send-mail`, and `git am` (an abbreviation for “apply mailbox”) can work directly with email files to convert them to/from Git commits. This is particularly useful for open source projects where everyone can access the Git repository but fewer people have write access to it. In this case, someone could send me an email that contains all the metadata of a commit using one of these commands. Nowadays, typically this is done with a GitHub pull request instead (which we'll cover in chapter 11).

Let's display some commits in an email-style format.

Problem

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

Solution

- 1 Change to the directory containing your repository; for example,
`cd /Users/mike/GitInPracticeRedux/`.
- 2 Run `git log --format=email --reverse --max-count 2` and, if necessary, `q` to exit. The output should resemble the following.

Listing 4.3 Output: email-formatted log

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

Commit author ② From 06b5eb58b62ba8bbbee258705c636ca7ac20b49 Mon Sep 17 00:00:00 2001 **Unix mailbox date** ①
 From: Mike McQuaid <mike@mikemcquaid.com>
Commit date ③ Date: Thu, 28 Nov 2013 15:49:30 +0000
 Subject: [PATCH] Remove unfavourable review file. **Commit subject** ④

```
From 36640a59af951a26e0793f8eb0f4cc8e4c030167 Mon Sep 17 00:00:00 2001
From: Mike McQuaid <mike@mikemcquaid.com>
Date: Thu, 28 Nov 2013 15:57:43 +0000
Subject: [PATCH] Ignore .tmp files.
```

- ① can be safely ignored. The first part is the SHA-1 hash for the commit. The log output is generated in the Unix *mbox* (short for mailbox) format. The second, date part is not affected by the commit date or contents but is a special value used to indicate that this was outputted from Git rather than taken from the real Unix mbox.
- ② is the author of the commit. This is one of the reasons Git stores a name and an email address for authors and in commits; it eases the transition to email format. A commit can be seen as an email sent by the author of the commit requesting that a change be made.
- ③ is the date on which the commit was made. This also sets the date for the email in its headers.
- ④ is the first line of the commit message prefixed with [PATCH]. This is another reason to structure your commit messages like emails (as mentioned in technique 4).

If there's more than one line in a commit message, the other lines are shown as the message body. Remember, if you use the `--patch` (or `-p`) argument, then the `git log` output will also include the changes made in the commit. With this argument provided, each outputted `git log` entry will contain the commit and all the metadata necessary to convert it to or from an email.

Discussion

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

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

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

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

Listing 4.4 Fuller log snippet

```
# git log --format=fuller
commit 334181a038e812050051776b69f0a80187abbeed
Author:      BrewTestBot <brew-test-bot@googlegroups.com>
AuthorDate:  Thu Jan 9 23:48:16 2014 +0000
Commit:     Mike McQuaid <mike@mikemcquaid.com>
CommitDate: Fri Jan 10 08:19:50 2014 +0000

    rust: add 0.9 bottle.

...
```

This snippet shows a single commit from Homebrew, an open-source project accessible at <https://github.com/Homebrew/homebrew>. This was used because in the `Git-InPracticeRedux` repository, all the previous commits will have the same author and committer, author date, and commit date.

WHY DO COMMITS HAVE AN AUTHOR AND A COMMITTER? The fuller commit output shows that for a commit, there are two recorded actions: the original author of the commit and the committer (the person who added this commit to the repository). These two attributes are both set at `git commit` time. If they're both set at once, then why are they separate values? Remember, you've seen repeatedly that commits are like emails and can be formatted as emails and sent to others. If I have a public repository on GitHub, other users can clone my repository but can't commit to it.

In these cases they may send me commits through a pull request (discussed later in section 10.1) or by email. If I want to include these in my repository, the separation between committing and authoring means I can then include these commits, and Git stores the person who, for example, made the code changes and the person who added these changes to the repository (hopefully after reviewing them). This means you can keep the original attribution for the person who did the work but still record the person who added the commit to the repository and (hopefully) reviewed it. This is particularly useful in open source software; with other tools, such as Subversion, if you don't have commit access to a repository, the best attribution you could hope for would be something like "Thanks to Mike McQuaid for this commit!" in the commit message.

In Subversion the equivalent `git blame` command is `svn blame`. It also has an alias called `svn praise`. In Git there's no such alias by default (but technique 50 will later show you how to create one yourself). I'm sure there's a joke to be made about the fact that Subversion offers praise and blame equally but Git offers only blame!

4.2 Custom output format

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

Let's create a more prose-like format for `git log`.

Listing 4.5 Custom prose log format

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

6 weeks ago Mike McQuaid did: Ignore .tmp files.
6 weeks ago Mike McQuaid did: Remove unfavourable review file.
6 weeks ago Mike McQuaid did: Add first review temporary file.
6 weeks ago Mike McQuaid did: Rename book file to first part file.
9 weeks ago Mike McQuaid did: Start Chapter 2.
3 months ago Mike McQuaid did: Joke rejected by editor!
3 months ago Mike McQuaid did: Improve joke comic timing.
3 months ago Mike McQuaid did: Add opening joke. Funny?
3 months ago Mike McQuaid did: Initial commit of book.
```

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

- `%ar` is the relative format date on which the commit was authored.
- `%an` is the name of the author of the commit.
- `did:` is text that's displayed the same in every commit and isn't a placeholder.
- `%s` is the commit message subject (the first line).

You can see the complete list of these placeholders in `git log --help`. There are too many for me to detail them all in this book. The large number of placeholders should mean you can customize `git log` output into almost any format.

4.3 Releasing logs: git shortlog

git shortlog shows the output of git log in a format that's typically used for open source software-release announcements. It displays commits grouped by author with one commit subject per line.

Listing 4.6 Output: short log

```
# git shortlog HEAD~6..HEAD
Mike McQuaid (9):
    Joke rejected by editor!
    Start Chapter 2.
    Rename book file to first part file.
    Add first review temporary file.
    Remove unfavourable review file.
    Ignore .tmp files.
```



The diagram shows two callouts. Callout 1, labeled '1 Commit author', points to the line 'Mike McQuaid (9):'. Callout 2, labeled '2 Commit message', points to the first line of the commit messages, 'Joke rejected by editor!'.

① shows the name of the author of the following commits and how many commits they've made.

② shows the first line of the commit message.

The commit range (HEAD~6..HEAD) is optional, but typically you'd want to use one to create a software-release announcement for any version after the first.

4.4 The ultimate log output

As mentioned previously, often the git log output is too verbose or doesn't display all the information you wish to query in a compact format. It's also not obvious from the output how local or remote branches relate to the output.

I have a selection of format options I refer to as my "ultimate log output." Let's look at the output with these options.

Listing 4.7 Output: graph log

```
# git log --oneline --graph --decorate
* 36640a5 (HEAD, origin/master, origin/HEAD, master) Ignore .tmp files.
* 06b5eb5 Remove unfavourable review file.
* fcd8f6e Add first review temporary file.
* c6eed66 Rename book file to first part file.
* ac14a50 Start Chapter 2.
* 07fc4c3 Joke rejected by editor!
* 85a5db1 Improve joke comic timing.
* 6b437c7 Add opening joke. Funny?
* 6576b68 Initial commit of book.
```

This output format displays each commit on a single line. The line begins with a branch graph indicator (which I'll explain shortly) followed by the short SHA-1

(which is useful for quickly copying and pasting), the branches, tags (introduced in technique 36), and HEAD (which points to this commit in parentheses), and ends with the commit subject.

As you may have noticed, this format is similar to that of the first two columns of GitX (see figure 4.1). The *GitInPracticeRedux* repository doesn't currently have any merge commits. Let's see what the graph log output looks like with some of them.

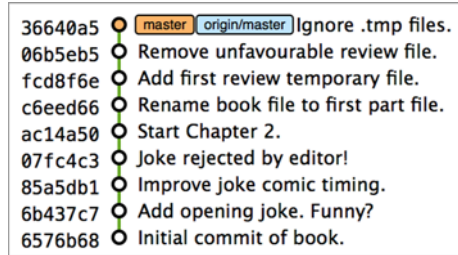


Figure 4.1 GitX graph output

Listing 4.8 Graph log merge commit snippet

```
# git log --oneline --graph --decorate
* 129cce6 (origin/master, origin/HEAD, master) Merge branch 'testing'
| \
| * a86067a (origin/testing, testing) testing branch commit
* | 1a36bd6 master branch commit
...
```

Here you can see the branch graph indicator becoming more useful. Like the graphical tools you saw in technique 4, this displays branch merges and the commits on different branches, using ASCII symbols to draw lines. The * means a commit that was made. Each line follows a single branch. Reading from the bottom up, you can see from the preceding listing that a commit was made on the master branch, a commit was made on the testing branch, and then the testing branch was merged into master. Both the testing and master branches remain (haven't been deleted), and both have been pushed to their respective remote branches. All this from just three lines of ASCII output. Hopefully you can see why I love this presentation. Typing `git log --oneline --graph --decorate` is unwieldy, so you'll see later in technique 50 how to shorten this to something like `git l` by using an alias.

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

I'm sure all developers have been in a situation where they've seen some line of code in a file and wondered why it is was written that way. As long as the file is stored in a Git repository, it's easy to query who made a change as well as when and why (given a good commit message was used) a certain change was made.

You could do this by using `git diff` or `git log --patch`, but neither of these tools is optimized for this particular use case; they both usually require reading through a lot of information you aren't interested in to find the information you want. Instead, let's see how to use a command designed specifically for this use case: `git blame`.

Problem

You wish to show the commit, person, and date on which each line of `GitInPractice.asciidoc` was changed.

Solution

- 1 Change to the directory containing your repository; for example,
`cd /Users/mike/GitInPracticeRedux/`.
- 2 Run `git blame --date=short 01-IntroducingGitInPractice.asciidoc`. The output should resemble the following.

Listing 4.9 Blame output

```
# git blame --date=short 01-IntroducingGitInPractice.asciidoc
^6576b68 GitInPractice.asciidoc (Mike McQuaid 2013-09-29 1)
= Git In Practice
6b437c77 GitInPractice.asciidoc (Mike McQuaid 2013-09-29 2)
== Chapter 1
07fc4c3c GitInPractice.asciidoc (Mike McQuaid 2013-10-11 3)
// TODO: think of funny first line that editor will approve.
ac14a504 GitInPractice.asciidoc (Mike McQuaid 2013-11-09 4)
== Chapter 2
ac14a504 GitInPractice.asciidoc (Mike McQuaid 2013-11-09 5)
// TODO: write two chapters
```

First, note that the output shows `GitInPractice.asciidoc` rather than `01-IntroducingGitInPractice.asciidoc`. This is because the filename has been changed since these changes were made. `git blame` is only showing changes to lines in the file and ignoring that the file was renamed. This is useful, because it means you don't lose all blame data whenever you rename a file.

The blame output shows the following:

- `--date=short` is used to display only the date (not the time). This accepts the same formats as the `--date` flag for `git log`. It was used in the preceding listing to make it more readable, because `git blame` lines tend to be very long.
- The `^` (caret) prefix on the first line indicates that this line was inserted in the initial commit.
- Each line contains the short SHA-1, filename (if the line was changed when the file had a different name), parenthesized name, date, line number, and line contents. For example, in commit `6b437c77` on September 29, 2013, Mike McQuaid added the `== Chapter 1` line to `GitInPractice.asciidoc` (although the file is now named `01-IntroducingGitInPractice.asciidoc`).

You've shown who changed each line of a file, in which commit, and when the commit was made.

Discussion

`git blame` has a `--show-email` (or `-e`) flag that can show the email address of the author instead of the name.

You can use the `-w` flag to ignore whitespace changes when finding where the line changes came from. Sometimes people fix stuff like indentation or whitespace on a line, which makes no functional difference to the code in most programming languages. In these cases, you want to ignore whitespace changes so you can look at the changes that affect program behavior.

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

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

Technique 32 Finding which commit caused a particular bug: `git bisect`

The only thing worse than finding a bug in software and having to fix it is having to fix the same bug multiple times. A bug that was found, fixed, and has appeared again is typically known as a *regression*.

The traditional workflow for finding regressions is fairly painful. You typically keep checking out older and older revisions in the version control history until you find a commit in which the bug wasn't present, check out newer and newer revisions until you find where it happens again, and repeat the process to narrow it down. It's a tedious exercise, which is made worse by your having to fix the same problem again.

Fortunately, Git has a useful tool that makes this process much easier for you: `git bisect`. It uses a binary search algorithm to identify the problematic commit as quickly as possible; it effectively automates the process of searching backward and forward through history that I explained earlier.

The `git bisect` command takes good and bad arguments that you use to tell it that a particular commit didn't have the bug (good) or did have the bug (bad). It assumes that the bug disappears and reappears multiple times but occurred once, so it can make the assumption that the commit that caused a particular bug is the first one chronologically that contains that bug. It uses this assumption, records the good and bad commits, and uses this information to narrow down the commits each time. For example, if it was bisecting between commits from Monday (good) to Friday (bad), then if a commit on Wednesday was known to be good, it could narrow the search to Monday, Tuesday, or Wednesday. This halving of the search space each time is known as a *binary search* because it makes a binary decision each time: was the bad commit before or after this one?

For a simple example, let's try to find out which commit renamed a particular file (without manually looking through the history).

Problem

You wish to locate the commit that renamed `GitInPractice.asciidoc` to `01-Introducing-GitInPractice.asciidoc`.

Solution

- 1 Change to the directory containing your repository; for example,
`cd /Users/mike/GitInPracticeRedux/`.
- 2 Run `git bisect start`. There will be no output.
- 3 Run `git bisect bad`. There will be no output.
- 4 Run `git bisect good 6576b6`, where `6576b6` is the SHA-1 of any commit that you know was before the rename. The output should resemble listing 4.1.
- 5 Check the names of the files in the directory by running `ls *.asciidoc*`.
- 6 When the `.asciidoc` file is named `GitInPractice.asciidoc`, run `git bisect good` to indicate that the file hasn't been renamed yet. When the `.asciidoc` file is named `01-IntroducingGitInPractice.asciidoc`, run `git bisect bad` to indicate that the file has been renamed. The output should be similar each time. No other parameters are required to `git bisect good` or `git bisect bad`; they automatically check out the next revision to be checked when they're run.
- 7 Eventually the first bad commit will be found. The output should resemble listing 4.2.
- 8 Run `git bisect reset`. The output should resemble listing 4.3.

Listing 4.10 Output: first good bisect

```
# git bisect good
Bisecting: 3 revisions left to test after this (roughly 2 steps)
[ac14a50465f37cfb038bdecd1293eb4c1d98a2ee] Start Chapter 2.
```

Steps remaining 1
New commit 2

- 1 shows how many revisions remain untested and, using the binary search algorithm, roughly how many more `git bisect` invocations remain until you find the problematic commit.
- 2 shows the new commit SHA-1 that `git bisect` has checked out for examining whether this commit is good (the bug isn't present) or bad (the bug is present).

Listing 4.11 Output: final bad bisect

```
# git bisect bad
c6eed6681efc8d0bff908e6dbb7d887c4b3fab3e is the first bad commit
commit c6eed6681efc8d0bff908e6dbb7d887c4b3fab3e
Author: Mike McQuaid <mike@mikemcquaid.com>
```

Bisect result 1
Commit information 2

```

Date:    Thu Nov 28 15:39:38 2013 +0000

    Rename book file to first part file.

:000000 100644 0000000000000000000000000000000000000000000000000000
5e02125ebbc8384e8217d4370251268e867f8f03 A
01-IntroducingGitInPractice.asciidoc
:100644 000000 5e02125ebbc8384e8217d4370251268e867f8f03
000000000000000000000000000000000000000000000000000 D
GitInPractice.asciidoc

```

- ① shows the commit that has been identified as causing the bug or, in this case, the rename. This matches the commit message here, so this is a slightly silly example, but typically it allows you to then examine these changes and identify what in this commit caused the regression.
- ② shows the `git show` information for this commit.
- ③ shows the old and new file mode and SHA-1 for the new object (after renaming).
- ④ shows the old and new file mode and SHA-1 for the old object (before renaming).

From figure 4.2, you can see that `git bisect` creates new, temporary refs (they're removed by `git bisect reset`) as it is working. These indicate the commits that were marked by `git bisect bad` and `git bisect good` while working through the history. The `refs/bisect/bad` ref points to the final, bad commit that was detected.

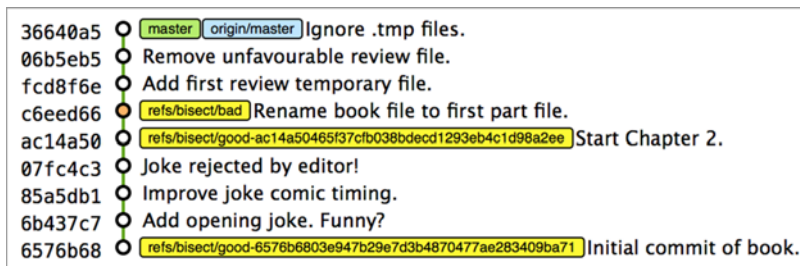


Figure 4.2 GitX bisect output before `git bisect reset`

You've located the commit that renamed `GitInPractice.asciidoc`.

Discussion

Each time `git bisect good`, `git bisect bad`, or `git bisect reset` is run, Git checks out the relevant next commit for examination. As a result, it's important to ensure that all outstanding changes have been committed (or stashed) before you use `git bisect`.

As you can see from table 4.1, as the number of commits increases, the maximum number of commits that need to be checked increases much more slowly. This algorithm means you can quickly navigate through a huge number of commits with `git bisect` without too many steps.

Table 4.1 Bisect binary search performance

Total commits	Max checked commits
10	6
100	13
1000	19

If you wish to examine the steps followed in a `git bisect` operation, you can run `git bisect log`.

Listing 4.12 Output: bisect log

```
# git bisect log
git bisect start
# bad: [36640a59af951a26e0793f8eb0f4cc8e4c030167]
  Ignore .tmp files.
git bisect bad 36640a59af951a26e0793f8eb0f4cc8e4c030167
# good: [6576b6803e947b29e7d3b4870477ae283409ba71]
  Initial commit of book.
git bisect good 6576b6803e947b29e7d3b4870477ae283409ba71
# good: [ac14a50465f37cfb038bdecd1293eb4c1d98a2ee]
  Start Chapter 2.
git bisect good ac14a50465f37cfb038bdecd1293eb4c1d98a2ee
# bad: [fcd8f6e957a03061cdf411851fe38034a44c97ab]
  Add first review temporary file.
git bisect bad fcd8f6e957a03061cdf411851fe38034a44c97ab
# bad: [c6eed6681efc8d0bff908e6dbb7d887c4b3fab3e]
  Rename book file to first part file.
git bisect bad c6eed6681efc8d0bff908e6dbb7d887c4b3fab3e
# first bad commit: [c6eed6681efc8d0bff908e6dbb7d887c4b3fab3e]
  Rename book file to first part file.
```

1 Bisect command

2 Commit SHA-I

3 Commit subject

4 Bisect result

- 1** shows the `git bisect` command invoked at this step.
- 2** shows the status and SHA-I of a commit.
- 3** shows the commit subject of a commit.
- 4** shows the final result of the entire bisect operation.

If you already know a bug came from particular files or paths in the working tree, you can specify these to `git bisect start`. For example, if you knew the changes that caused the regression were in the `src/gui` directory, you could run `git bisect start src/gui`. This means only the commits that changed the contents of this directory will be checked, which makes things even faster.

If it's difficult or impossible to tell whether a particular commit is good or bad, you can run `git bisect skip`, which will ignore it. Given enough other commits, `git bisect` will use another to narrow the search instead.

4.5 Automating git bisect

git bisect is already useful, but wouldn't it be even better if, rather than having to keep typing git bisect good or git bisect bad, it could run automatically and tell you which commit caused the regression? This is possible with git bisect run.

git bisect run is run instead of git bisect good or git bisect bad (after a git bisect start, git bisect good, or git bisect bad, and before a git bisect reset) and automates future runs of git bisect good and git bisect bad. It uses the exit code of a process to identify whether the command was successful. For example, if you run the command `ls GitInPractice.asciidoc`, it returns an exit code of 0 on success (when the file is present) and 1 on failure (when the file is not). Let's take advantage of this to use it with git bisect run.

Listing 4.13 Output: bisect run

```
# git bisect start

# git bisect bad

# git bisect good 6576b6

Bisecting: 3 revisions left to test after this (roughly 2 steps)
[ac14a50465f37cfb038bdec1293eb4c1d98a2ee] Start Chapter 2.

# git bisect run ls GitInPractice.asciidoc

Bisecting: 3 revisions left to test after this (roughly 2 steps)
[ac14a50465f37cfb038bdec1293eb4c1d98a2ee]
Start Chapter 2.
running ls GitInPractice.asciidoc
GitInPractice.asciidoc
Bisecting: 1 revision left to test after this (roughly 1 step)
[fcd8f6e957a03061cdf411851fe38034a44c97ab]
Add first review temporary file.
running ls GitInPractice.asciidoc
ls: GitInPractice.asciidoc: No such file or directory
Bisecting: 0 revisions left to test after this (roughly 0 steps)
[c6eed6681efc8d0bff908e6dbb7d887c4b3fab3e]
Rename book file to first part file.
running ls GitInPractice.asciidoc
ls: GitInPractice.asciidoc: No such file or directory
c6eed6681efc8d0bff908e6dbb7d887c4b3fab3e is the first bad commit
commit c6eed6681efc8d0bff908e6dbb7d887c4b3fab3e
Author: Mike McQuaid <mike@mikemcquaid.com>
Date: Thu Nov 28 15:39:38 2013 +0000

    Rename book file to first part file.

:000000 100644 000000000000000000000000000000000000000000000000
5e02125ebbc8384e8217d4370251268e867f8f03 A
01-IntroducingGitInPractice.asciidoc
:100644 000000 5e02125ebbc8384e8217d4370251268e867f8f03
000000000000000000000000000000000000000000000000 D
GitInPractice.asciidoc
bisect run success
```

The output is identical to the `git bisect log` output or the combined output of all the other `git bisect` operations. No human intervention was required in the preceding output; it ran until it reached a result.

A typical case would be writing a unit test that reproduces a regression and using that with `git bisect run` to quickly test a large number of commits.

HOW CAN YOU STOP GIT BISECT FROM OVERWRITING YOUR TEST? Because `git bisect good` and `git bisect bad` perform a `git checkout` each time, you need to make sure the regression test isn't overwritten by other files or committed after the earliest bad commit. The easiest way to do this is to make a copy of the test in another directory outside the Git working directory, so `git bisect run` won't change its contents as it checks out different commits.

4.6 Summary

In this chapter, you learned the following:

- How to filter `git log` output by author, date, commit message, and merge commits
- How to display only a single commit or a requested number of commits
- How to display `git log` output in various formats
- How to display commits in an open source release-announcement format
- How to display branching effectively with `git log`
- How to show who changed each line of a file, when, and why, and the original filename
- How to use `git bisect` to search quickly (but manually) through the history with `git bisect good` and `git bisect bad`, to identify regressions
- How to use `git bisect run` to search automatically through the history to identify regressions with a test

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.

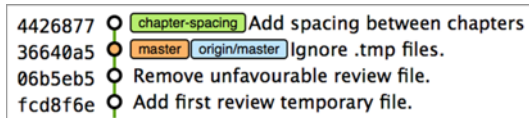


Figure 5.1 Local repository before merge without a merge commit

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 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.

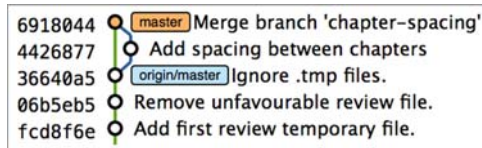


Figure 5.2 Local repository after `git merge --no-ff chapter-spacing`

`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-IntroducingGitInPractice.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.

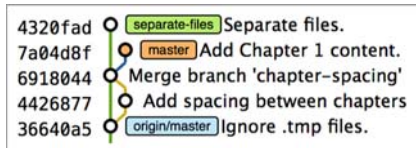


Figure 5.3 Local repository before merge-conflict resolution

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
=====
// TODO: think of funny first line that editor will approve.
>>>>>> separate-files ④ Branch separator
⑤ Current version
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
```

- ❶ 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.
- ❷ 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.
- ❸ shows a line that was deleted in the incoming (*separate-files*) branch's commit(s). The `-` is in the first column as a result.
- ❹ shows a line that was deleted in the current (*master*) branch's commit(s). The `-` is in the second column as a result.
- ❺ 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.

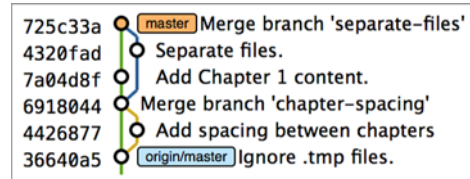


Figure 5.4 Local repository after merge-conflict resolution

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.

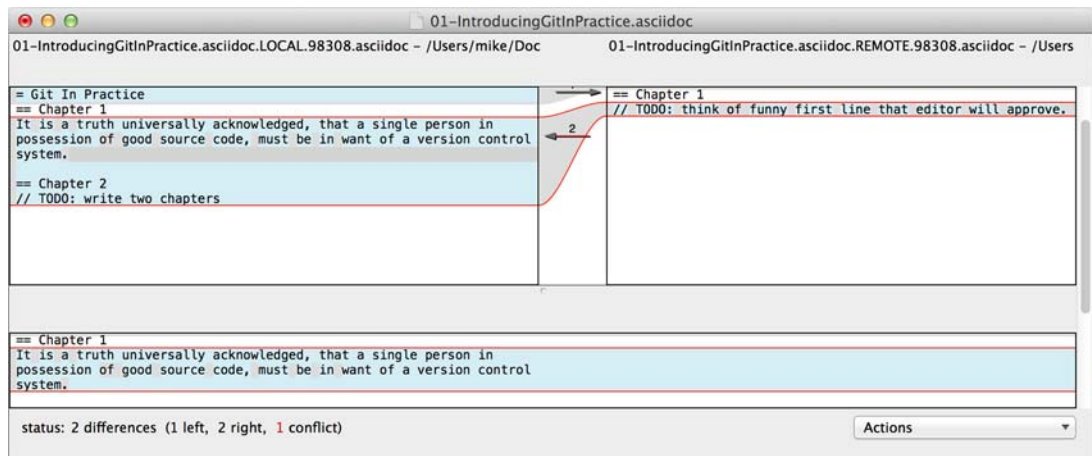


Figure 5.5 `opendiff` merge-conflict resolution

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.

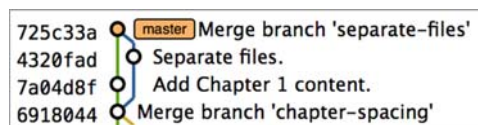


Figure 5.6 Local repository before git tag

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`.

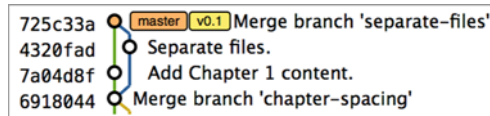


Figure 5.7 Local repository after `git tag`

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 Droppable
commit

2 Kept commit

- 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

Rewriting history and disaster recovery

This chapter covers

- Viewing the history of all changes made to branches over time
- Making a branch point to a previous commit
- Changing the parent of commits to point to another
- Forcing push-rewritten history to a remote repository
- Rewriting the entire history of a branch
- Avoiding losing your work

Technique 4 briefly discussed that Git is capable of rewriting the history of a repository. Because each repository contains the entire history, this ranges from undoing a single commit to rewriting data on every commit in the repository. I make use of rewriting history regularly to ensure that merged branches have a clean, readable history made up of small commits before merging. You can read more about this in section 13.2 and chapter 14. Before jumping in to rewriting history, let's start by learning about how to avoid losing committed data using `git reflog`.

Technique 41 Listing all changes including history rewrites: git reflog

Recall from technique 3 that each commit points to the previous (parent) commit and that this is repeated all the way to the branch's initial commit. As a result, the branch pointer can be used to reference not just the current commit it points to, but also every previous commit. If a previous commit changes in any way, then its SHA-1 hash changes too. This in turn affects the SHA-1 of all its descendant commits. This provides good protection in Git from accidentally rewriting history, because all commits effectively provide a checksum of the entire branch up until this point.

Git's *reflog* (or *reference log*) is updated whenever a commit pointer is updated (like a HEAD pointer or branch pointer). This includes previous actions you've seen that don't involve rewriting history, such as committing and changing branches. Let's see the reflog contents for your previous actions on the repository.

Problem

You wish to view the state of the reflog for the HEAD pointer.

Solution

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

Listing 6.1 Reflog output

```
# git reflog
```

```
3e3c417 HEAD@{0}: checkout: moving from v0.1-release to master
a8200e1 HEAD@{1}: commit: Add release preface.
dfe2377 HEAD@{2}: checkout: moving from master to v0.1-release
3e3c417 HEAD@{3}: revert: Revert "Advanced practice technique."
...
```

New
commit

2

1 Change
of branch

3 Revert
commit

- 1 shows that you called `git checkout` to move from the `v0.1-release` branch to the `master` branch. This was the most recent action performed, and it changed the HEAD pointer.
- 2 shows that you used `git commit` to create a new commit in the repository with the subject `Add release preface`.
- 3 shows that you reverted a previous commit, which created a new commit with the message `Revert "Advanced practice technique."`. Note that this is the same SHA-1 as 1, because they both point to the latest commit on the master branch.

Let's compare this to the current GitX output. You can see in figure 6.1 that the SHA-1s match those in the `git reflog` output:

- The latest commit on the `v0.1-release` branch (`a8200e1, HEAD@{1}`)
- The penultimate commit on the `v0.1-release` branch (`dfe2377, HEAD@{2}`)
- The latest commit on the `master` branch (`3e3c417, HEAD@{0}, HEAD@{3}`)

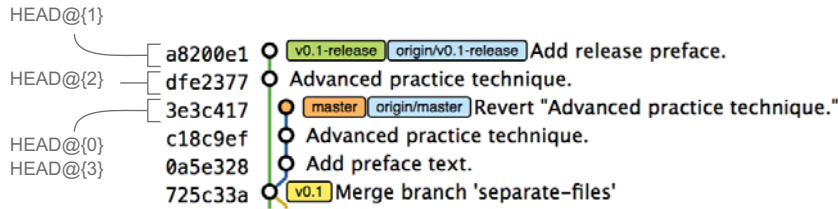


Figure 6.1 GitX reflog comparison output

Recall that in section 1.7 you learned some different types of refs. `HEAD@{0}` is another type of ref and can be used with commands such as `git log` and `git cherry-pick` to view the state based on that commit. I always prefer to use the SHA-1, because SHA-1s for particular commits never change (even with the history rewriting you’ll learn later this chapter), but `HEAD@{0}` for example will change with any action that affects the `HEAD` pointer.

Remember `git stash` from technique 23? When you use the `git stash` commands, they don’t appear in the `git reflog` output. Again, for this reason, once you learn about rewriting history in this chapter, I’d recommend making commits and rewriting them later rather than relying too much on `git stash`.

You have viewed the changing state of the `HEAD` pointer by viewing the reflog.

Discussion

Running `git reflog` is an alias for `git log --walk-reflogs --abbrev-commit --pretty=oneline`. `git reflog` can also take all `git log` flags such as `--max-count` and `--patch`. You can see the `git log` formatting in technique 30.

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

ARE REFLOGS SHARED BETWEEN REPOSITORIES? Reflogs are per repository. They aren’t shared with other repositories when you `git push` and aren’t fetched when you `git fetch`. As a result, they can only be used to see actions that were made in the Git repository on your local machine. Bear this in mind when you’re rewriting history: you can easily view the previous state on your current machine, but not that from other machines.

6.1 Avoiding and recovering from disasters

Despite the scary phrase “rewriting history,” you’ll see throughout this chapter that any operation that acts on commits (such as `git rebase`) rather than the working directory (such as `git reset --hard` with uncommitted changes) is easily recoverable using `git reflog` for 90 days after the changes were made.

The main rule to avoid data loss therefore is *commit early and commit often*. Now that you know how to rewrite history, you should think of committing not as a complex operation but similar to a `Save` operation in most other pieces of software. Commit

whenever you've written anything useful that you don't want to lose, and then rewrite your history later into small, readable commits.

The easiest (and most common) way to lose data with Git is when it hasn't been committed and you accidentally run `git reset --hard` or `git checkout --force`, and it's overwritten on disk. This can be somewhat avoided by having regular backups of your repository while you work (such as using Time Machine on OS X), but it's generally better to let Git handle this for you by committing more often.

Another way to secure your data with Git is to regularly push to remote work branches that you've agreed nobody else will commit to. If you've agreed that no one else will commit to these work branches, it's reasonable to rewrite and force-push to them in the same way you might rewrite a local branch. This means these changes will be safe on the remote repository and downloaded by anyone else's `git fetch` from this repository. This is useful in case there is a hardware failure on your machine; you can get back the data from the branch on the remote repository.

If things ever go really badly and you suffer disk corruption with important but unpushed commits in your repository, you can run the `git fsck` tool. It verifies the integrity of the repository and prints out any missing or corrupt objects that it finds. You can then remove these corrupt objects, restore them from backups, or check whether other users of the same repository have the same objects. Hopefully the corrupted objects aren't those with the most recent work you wish to recover.

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

In technique 39 you learned how to use `git revert` to revert a commit that made changes you want to undo. I stated previously that rewriting history is a bad practice if the commit to be reverted has already been pushed publicly (I'll elaborate more on this later in this chapter). But what if the commit hasn't been pushed yet? In this case you can use a command you first saw in technique 19: `git reset`.

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

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

Let's create a temporary commit (that hasn't been pushed) that you can reset:

- 1 Change to the directory containing your repository: for example,
`cd /Users/mike/GitInPracticeRedux/`.
- 2 Run `git checkout master`.
- 3 Edit `00-Preface.asciidoc`, and make a change to the file.
- 4 Run `git commit --message="Update preface." 00-Preface.asciidoc`. The output should resemble the following.

Listing 6.2 Output: commit to be reset

```
# git commit --message="Update preface." 00-Preface.asciidoc

[master 4455fa9] Update preface.
1 file changed, 1 insertion(+), 1 deletion(-)
```

In figure 6.2, you can see the state of GitX after the `git commit`.

```
4455fa9 ● master Update preface.
3e3c417 ● origin/master Revert "Advanced practice technique."
c18c9ef ● Advanced practice technique.
```

Figure 6.2 Commit to be reset in GitX

In this case, let's try resetting to the previous commit on the same branch; this is an alternative to using `git revert`.

Problem

You wish to undo the last commit on 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 reset HEAD^`. The output should resemble the following.

Listing 6.3 Output: resetting a commit

```
# git reset HEAD^
```

```
Unstaged changes after reset:
```

```
M    00-Preface.asciidoc
```

← ① **Status message**

← ② **Uncommitted changes**

- ① shows that undoing this commit has left the file modified, but its modifications haven't yet been added to the index. This could be done later with `git add`.
- ② shows that the `00-Preface.asciidoc` file is currently in a modified state after the reset operation.

Figure 6.3 shows the state of GitX after the `git reset`. The commit that was created by the `git commit` command has disappeared from GitX.

```
3e3c417 ● master origin/master Revert "Advanced practice technique."
c18c9ef ● Advanced practice technique.
```

Figure 6.3 GitX after resetting a commit

You have reset the master branch pointer to point to a previous commit.

Discussion

Remember when I said in technique 41 that `git reflog` is useful in avoiding the loss of commits? Let's imagine that you reset the previous commit but later realize this was a mistake. Let's run `git reflog` and see if you can get anything useful from the output.

Listing 6.4 Reflog output after resetting a commit

```
# git reflog HEAD
3e3c417 HEAD@{0}: reset: moving to HEAD^
4455fa9 HEAD@{1}: commit: Update preface.
3e3c417 HEAD@{2}: checkout: moving from v0.1-release to master
a8200e1 HEAD@{3}: commit: Add release preface.
...
```

- ❶ shows the commit-reset operation reset the state of the master branch to that before this commit. The SHA-1 (3e3c417) matches that of the checkout before this commit was made.
- ❷ shows the new commit that was made and then reset. The SHA-1 matches that output from the previous `git commit` command.

`reflog` has kept the record that this reset was made and the SHA-1s at each stage in this process. Let's use the SHA-1 output by ❷ from the `reflog` (and the previous `git commit` command) to restore this commit again:

- 1 Change to the directory containing your repository: for example, `cd /Users/mike/GitInPracticeRedux/`.
- 2 Run `git checkout master`.
- 3 Run `git reset 4455fa9`. There will be no output.

If you now examine figure 6.4, you'll see that the commit has been restored and the state is exactly the same as when it was made. The only record that the reset was ever made is now in the `reflog`.

Now run `git push` to send the commit to the remote repository.

```
4455fa9 ○ master Update preface.
3e3c417 ○ origin/master Revert "Advanced practice technique."
c18c9ef ○ Advanced practice technique.
```

Figure 6.4 Restored commit in GitX

WHEN ARE COMMITS REMOVED FROM THE REFLOG? Commits in the `reflog` that are older than 90 days and not ancestors of any other, newer commit in the `reflog` are removed by the `git gc` command. `git gc` can be run manually, but it never needs to be because it's run periodically by commands such as `git fetch`. In short, when you've removed a commit from all branches, you have 90 days to recover the data before Git will destroy it. In my experience, this is more than enough time; typically if I haven't remembered that I accidentally removed a commit within a few days, I never will.

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

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

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

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

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

Recall from technique 13 that rebasing is similar to merging but requires rewriting history. Let's create a branch that you can rebase:

- 1 Change to the directory containing your repository: for example,
`cd /Users/mike/GitInPracticeRedux/`.
- 2 Run `git checkout -b inspiration v0.1`.
- 3 Edit 01-IntroducingGitInPractice.asciidoc, and make a change to the file.
- 4 Run `git commit --message="Add Chapter 1 inspiration." 01-Introducing-GitInPractice.asciidoc`. The output should resemble the following.

Listing 6.5 Output: commit to be rebased

```
# git commit --message="Add Chapter 1 inspiration."
01-IntroducingGitInPractice.asciidoc

[inspiration 88e8b4b] Add Chapter 1 inspiration.
1 file changed, 1 insertion(+)
```

Figure 6.5 shows the new inspiration branch. It has a single commit, and the parent of that commit is the commit that has the v0.1 tag.

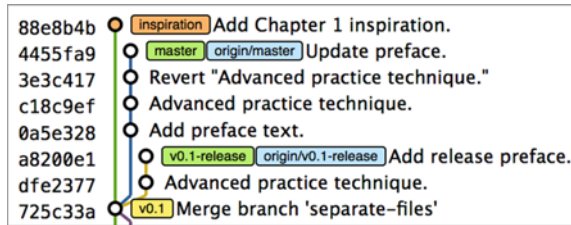


Figure 6.5 Newly created inspiration branch

Now let's rebase this branch.

Problem

You wish to rebase the inspiration branch on top of the v0.1-release branch.

Solution

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

Listing 6.6 Rebase output

```
# git rebase v0.1-release
```

```
First, rewinding head to replay your work on top of it...  ← ❶ HEAD rewind
Applying: Add Chapter 1 inspiration.                      ← ❷ Applying a commit
```

- ❶ shows that Git is moving the HEAD pointer to the latest commit on the v0.1-release branch. It's doing this so it can apply the newly created commit on the inspiration branch with the latest commit on the v0.1-release branch as its parent.
- ❷ shows a list of each commit (in this case, only one) re-created on the branch. Effectively, each commit on the branch being rebased is cherry-picked (discussed in technique 38) on top of the new base: the latest commit on the v0.1-release branch. Because their parent commits have changed, so do the SHA-1s of all the commits.

Figure 6.6 shows the rebased inspiration branch. It still has a single commit, but that commit's parent is now the latest commit on the v0.1-release branch rather than the commit tagged v0.1. Note that GitX reflowed and recolored some of the

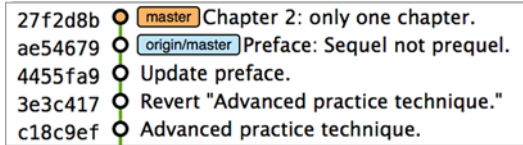


Figure 6.6 Rebased inspiration branch

branches; master is now to the right of v0.1-release, and v0.1-release is now green. This doesn't have any significance beyond GitX trying to make the output more readable.

Note that some editions of this book are printed in grayscale, so these colors may not be visible. Instead, please compare them to GitX on your computer.

You have rebased the inspiration branch on top of the v0.1-release branch.

Discussion

The argument to `git rebase` can be any ref. You could rebase on an arbitrary commit, but this is generally a bad idea. You should usually rebase on top of either an updated branch or a different branch/tag.

If you made multiple commits to the wrong branch, you can't use `git rebase` as is to fix this. But you can do so with `git rebase --interactive`, which you'll see in technique 44.

Let's look at the reflog again to see what effects the rebase had.

Listing 6.7 Reflog output after rebase

```
# git reflog
1 5d4ad83 HEAD@{0}: rebase finished: returning to refs/heads/inspiration
2 5d4ad83 HEAD@{1}: rebase: Add Chapter 1 inspiration.
3 a8200e1 HEAD@{2}: rebase: checkout v0.1-release
4 88e8b4b HEAD@{3}: commit: Add Chapter 1 inspiration.
5 725c33a HEAD@{4}: checkout: moving from master to inspiration
6 4455fa9 HEAD@{5}: reset: moving to 4455fa9
7 3e3c417 HEAD@{6}: reset: moving to HEAD^
...
```

Rebase completion 1 →

Rebase checkout 3 →

Rebased commit 2 ←

Commit pre-rebase 4 ←

1 shows that the rebase operation has completed successfully, so the inspiration branch was updated to point to the rebased commit.

2 shows the new commit that was created with the parent pointing to the latest commit on the v0.1-release branch. The inspiration branch was updated after this commit was successfully created. This avoids a situation where a failed rebase operation leaves a branch in an inconsistent state.

3 shows the beginning of the rebase operation by checking out the v0.1-release branch that is being used as a new parent.

4 shows the new commit that was made before it was rebased.

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

Sometimes git rebase operations may fail in a way similar to a git merge or git cherry-pick operation. There may be a merge conflict where changes have been made to the same parts of the same files that have been modified in rebased commits. The main difference when resolving a git rebase (or git cherry-pick) conflict is that, because there's no merge commit, it has to be done for each commit at a time.

If the preceding rebase had failed, the output would look something like this.

Listing 6.8 Output: rebase conflict

```
First, rewinding head to replay your work on top of it...
Applying: Add Chapter 1 inspiration.
Using index info to reconstruct a base tree...
M    01-IntroducingGitInPractice.asciidoc
Falling back to patching base and 3-way merge...
Auto-merging 01-IntroducingGitInPractice.asciidoc
CONFLICT (content): Merge conflict in
  01-IntroducingGitInPractice.asciidoc
Failed to merge in the changes.
Patch failed at 0001 Add Chapter 1 inspiration.
The copy of the patch that failed is found in:
  /Users/mike/Documents/GitInPracticeRedux/.git/rebase-apply/patch

When you have resolved this problem, run "git rebase --continue".
If you prefer to skip this patch, run "git rebase --skip" instead.
To check out the original branch and stop rebasing, run "git rebase
--abort".
```

1 Rebase begins

2 Merge conflict

3 Rebase instructions

- 1** shows the same first two lines as a successful rebase; the HEAD was rewound, and Git tried to apply the changes in the commit. The only difference is that, in this case, the changes couldn't be merged automatically.
- 2** shows the attempt by rebase to merge the multiple changes that were made to the same file. This may be successful, but in this case the merge failed, so rebase tells the user to solve it manually.
- 3** shows the instructions involved in solving the rebase conflict. There are three suggested flags:
 - `git rebase --continue` should be run after the normal merge conflict-resolution process of manually resolving the conflicts and marking them as fixed using `git add`. This continues the rebase operation by rebasing any further commits and, if successful, updating the rebased branch.
 - `git rebase --skip` means that, rather than solving the merge conflicts in this particular commit, the commit is skipped and the next one is applied instead. This may make sense in certain situations where the functionality of this commit has already been made by another commit on the branch you're rebasing on top of, making this commit redundant.
 - `git rebase --abort` gives up on the git rebase process altogether and returns the branch to its state before the rebase was attempted.

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

You may have thought to yourself that, given the various reset, cherry-pick, and commit skip options that you've seen in this chapter and the last, it would be nice if you could somehow combine them to alter the history of a branch into the form you'd like before pushing it elsewhere. Git provides a useful tool for this use case: the `--interactive` (or `-i`) flag for rebase.

For this example, you want to see how empty commits are handled. Normally you'd rarely do this; it's just to demonstrate how rebase handles them. This can happen after resolving conflicts so that a commit no longer has any changes. You might do this manually if you wanted to trigger an event in a system that monitors your Git repository and triggers events on each new commit, but without having to change any files. Let's create an empty commit on the inspiration branch:

- 1 Change to the directory containing your repository: for example,
`cd /Users/mike/GitInPracticeRedux/`.
- 2 Run `git checkout inspiration`.
- 3 Run `git commit --allow-empty --message "Empty commit"` to create an empty commit. The output should resemble the following:

```
# git commit --allow-empty --message "Empty commit"
[inspiration 26596ad] Empty commit
```

Problem

You wish to interactively rebase the history of a branch.

Solution

- 1 Change to the directory containing your repository: for example,
`cd /Users/mike/GitInPracticeRedux/`.
- 2 Run `git checkout inspiration`.
- 3 Run `git rebase --interactive v0.1`. An editor will appear, and the contents should resemble the following.

Listing 6.9 Interactive rebase `git-rebase-todo` file

```
pick dfe2377 Advanced practice technique.
pick a820e1 Add release preface.
pick 5d4ad83 Add Chapter 1 inspiration.
# pick 26596ad Empty commit
# Rebase 725c33a..5d4ad83 onto 725c33a
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
```

Inspiration commit 2

pick command 5

1 V0.1 release

3 Empty commit

4 Rebase range

6 reword command

7 edit command

```

# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out

```

8 squash command

9 fixup command

10 Rebase begins

- 1** shows the last commit on the v0.1-release branch. This appears here because you're rebasing back on top of the v0.1 tagged commit, so the rebase operation tries to rebase any commit ancestors on the inspiration branch that aren't ancestors of the v0.1 tagged commit. It may be undesirable to have these commits on your new branch; but, as mentioned in the file, if you remove this line from the file, the commit will be removed from the rebase. This is effectively the same as if there was a conflict and you used `git rebase --skip` to skip this commit. Even more powerfully, if you cut a line from here and perform another `git rebase --interactive` operation, as long as the commit still exists (it hasn't been purged due to being detached from any branch for 90 days), you can paste it back in, and rebase will include it in this operation.
- 2** shows the last commit that was made on the inspiration branch. This has the same commit contents (although a different SHA-1) as the original commit that you rebased on top of the v0.1-release branch.
- 3** shows a commit with no changes. These are commented out with a `#` character. You should never keep empty commits, so rebase is removing them for you automatically.
- 4** shows the range of commits that are being rebased (725c33a..5d4ad83) and what commit they're being rebased onto. In this case, the 725c33a commit is the commit tagged v0.1.
- 5** is one of several *rebase commands* that can be selected for each commit in an interactive rebase. The rebase runs through the list of commits from top to bottom and follows the command for each listed commit (skipping any that have been removed). The `pick` command means the commit should be included in the rebase as is. If this file is saved and closed without modification, every commit is picked, and the `git rebase --interactive` operation is identical to what the `git rebase` operation would be.
- 6** is a rebase command that, when it reaches that commit and is ready to apply it, opens a file in an editor, allowing you to customize the commit message. This is particularly useful when you later realize that a commit message was poor and you wish to change it before pushing.

- 7 behaves as if there were a merge conflict. Before the commit is applied, you're dropped into a terminal with `--continue/--skip/--abort` options, and you can edit the contents of the commit before proceeding. This is useful when you want to slightly modify a commit, perhaps so a later one can apply more cleanly or to change the way you solved a problem in a particular commit.
- 8 merges two or more commits into one. If a commit is marked to be squashed, the preceding commit (or commits if the previous commit was also marked to be squashed) won't be applied until the last adjacent squashed commit is reached. This last commit will then prompt for a commit message (like `edit`), and the default commit message will be a combination of all the commit messages for the commits that have been squashed together. This is useful for combining multiple commits; perhaps the first commit was missing a file, which was then added in the second commit. The commits can be squashed, which means when they're pushed to the remote repository, no one ever needs to know that a mistake was made when creating the first commit.
- 9 is very similar to the `squash` command, but it doesn't prompt for a message and instead uses the commit message of the first commit. This is used in situations where you want to combine commits but don't need to change commit messages.
- 10 is different than the previous commands in that it doesn't operate on a commit. Instead, the rest of the line after `exec` is run at that stage in the rebase process. This can be used to print debugging output while performing a rebase operation.

Edit and save the file so the only uncommented lines are the following.

Listing 6.10 Rebase reorder and fixup

```
pick 5d4ad83 Add Chapter 1 inspiration.
p dfe2377 Advanced practice technique.
f a8200e1 Add release preface.
# ...
```

The output should resemble the following.

Listing 6.11 Output: interactive rebase

```
# git rebase --interactive v0.1

[detached HEAD 0109344] Advanced practice technique. ← 1 Fixup commit
2 files changed, 2 insertions(+), 1 deletion(-)
Successfully rebased and updated refs/heads/inspiration. ← 3 Rebase success
```

Fixup
diff
2

- 1 shows the new commit that was created by the `fixup` command. It has the commit message of the first of the two commits, but it contains the changes from both commits.
- 2 shows the diffstat for the new commit created by the `fixup` command.
- 3 shows the successful result of the rebase operation.

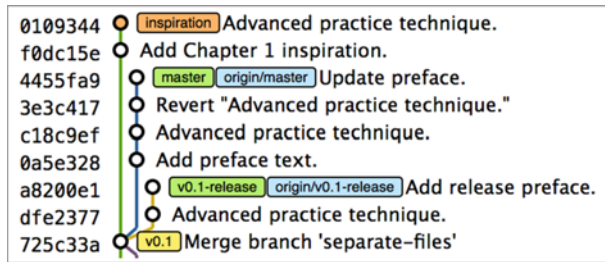


Figure 6.7 Interactively rebased inspiration branch

Figure 6.7 shows the state of the inspiration branch after the interactive rebase. Rather than being how it was before your first `git rebase`, it now has two commits, the latter of which was previously the prior, and it contains the contents of two commits.

You have successfully interactively rebased the inspiration branch on the `v0.1` tagged commit. Now push it using `git push --set-upstream origin inspiration`.

Discussion

Although complex, `git rebase --interactive` allows for powerful workflows (some of which are discussed later in this book). I typically always use an interactive rebase before I push a branch upstream; it allows me to take stock and consider what I want the history to look like. The factors I consider are whether any commits are now redundant or only cleaning up previous commits, whether any commit messages can be improved, whether any commits need to be reordered to make more sense, and whether any commits need to be removed or moved to other branches. `git rebase --interactive` allows me to do this for all my commits in an ordered process, so it works as a particularly effective review and modification tool.

Part 3 will cover some workflows, in which `git rebase --interactive` is a key part.

Technique 45 Pulling a branch and rebasing commits: *git pull --rebase*

Rebasing is often useful when you're pulling commits into your current branch. You almost certainly don't want to create a merge commit just because you've made commits on your current branch and want to fetch new commits from upstream. A merge commit will be created, however, if you've committed on this branch and pull in new commits. Instead of creating a merge conflict, you can use `git pull --rebase`.

To test `git pull --rebase`, let's create another clone of the same repository, make a new commit, and `git push` it. This will let you download new changes with `git pull --rebase` on the original remote repository.

Here are the steps to create another cloned, local repository and push a commit from it:

- 1 Change to the directory where you want the new `GitInPracticeRedux` repository to be created: for example, `cd /Users/mike/` to create the new local repository in `/Users/mike/GitInPracticeReduxPullTest`.

- 2 Run `git clone https://github.com/GitInPractice/GitInPracticeRedux.git` `GitInPracticeReduxPullTest` to clone into the `GitInPracticeReduxPullTest` directory.
- 3 Change directory to the new Git repository: in my case, `cd /Users/mike/GitInPracticeReduxPullTest/`.
- 4 Modify the `00-Preface.asciidoc` file.
- 5 Run `git commit --message 'Preface: Sequel not prequel.'`
`00-Preface.asciidoc`.
- 6 Run `git push`.

Now let's create a commit in your main, local repository:

- 1 Change to the directory containing your repository: for example,
`cd /Users/mike/GitInPracticeRedux/`.
- 2 Run `git checkout master`.
- 3 Edit `02-AdvancedGitInPractice.asciidoc`, and make a change to the file.
- 4 Run `git commit --message="Chapter 2: only one chapter." 02-AdvancedGitInPractice.asciidoc`. The output should resemble the following.

Listing 6.12 Output: commit to be reset

```
# git commit --message="Chapter 2: only one chapter."
02-AdvancedGitInPractice.asciidoc

[master 357d7db] Chapter 2: only one chapter.
1 file changed, 1 insertion(+), 1 deletion(-)
```

Figure 6.8 shows the state of the master branch before the `git pull --rebase` operation. Now let's perform a pull with a rebase.

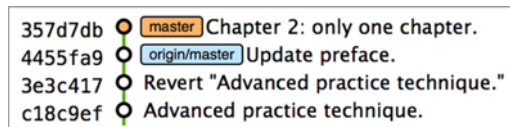


Figure 6.8 Commit before pull rebase

Problem

You want to pull commits from `origin/master` and rebase your current commits in `master` on top of the upstream changes.

Solution

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

Listing 6.13 Output: rebase pull

```
# git pull --rebase

remote: Counting objects: 3, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From https://github.com/GitInPractice/GitInPracticeRedux
 4455fa9..ae54679  master    -> origin/master
First, rewinding head to replay your work on top of it...
Applying: Chapter 2: only one chapter.
```

← ❶ **Fetch output**

← ❷ **Rebase output**

Recall that `git pull` is equivalent to running `git fetch && git merge`, and `git pull --rebase` is equivalent to running `git fetch && git rebase`.

- ❶ shows the output of the fetch operation. This is the same as if you'd run `git fetch`.
- ❷ shows the output of the successful rebase operation. The one commit that had already been made on your local master branch is rebased on top of the latest commit in the `origin/master` remote branch. This is the same as if you had run `git rebase origin/master` after `git fetch`.

Figure 6.9 shows the state of the master branch after the `git pull --rebase` operation. You can see that there's a new commit from `origin/master` (`ae54679`) and that the previous top commit on the local master branch has been rebased on top of this and has a new SHA-1 (`27f2d8b`). This works identically if there are multiple commits that need to be rebased.

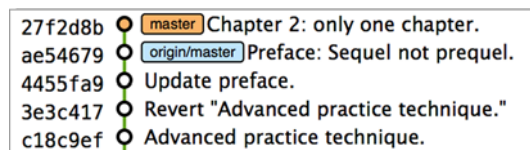


Figure 6.9 Commit after pull rebase

You've pulled with a rebase. Now `git push` to send these commits upstream.

Discussion

`git pull --rebase` is sometimes recommended as a sensible default to use instead of `git pull`. You'll rarely want to create a merge commit on a `git pull` operation, so using `git pull --rebase` guarantees that this won't happen. This means when you do push this branch, it will have a simpler, cleaner history. Once you understand how to rebase and solve conflicts, I recommend using `git pull --rebase` by default.

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

If you modify history on a branch and then try to perform a `git push` operation, it will fail. This is to stop you from accidentally writing remote history that other users are relying on. It's possible to do this, but you need to be more explicit in your syntax to indicate that you're aware you're performing a dangerous operation.

Let's try to rebase the inspiration branch again and push it:

- 1 Change to the directory containing your repository: for example, `cd /Users/mike/GitInPracticeRedux/`.
- 2 Run `git checkout inspiration`.
- 3 Run `git push` to ensure that all the changes are up to date.
- 4 Run `git rebase v0.1-release`.
- 5 Run `git push` again. The output should resemble the following.

Listing 6.14 Output: rewritten history push failure

```
# git push
To https://github.com/GitInPractice/GitInPracticeRedux.git
! [rejected]        inspiration -> inspiration (non-fast-forward)
error: failed to push some refs to
'https://github.com/GitInPractice/GitInPracticeRedux.git'
hint: Updates were rejected because the tip of your current branch
is behind
hint: its remote counterpart. Integrate the remote changes (e.g.
hint: 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help'
for details.
```

Push failure ①

Local/remote branches ②

① shows the local inspiration branch that you attempted to push to the remote inspiration branch. Unfortunately, this request was rejected because it was a non-fast-forward—it wouldn't be advancing the current history but instead rewriting it.

② shows the error message from `git push`. It fails because the branch you're pushing lacks changes from the branches you're pushing to. This is because it's not easily possible for the remote repository to know whether you have commits on that branch that you need to fetch before pushing or whether you've modified the existing history of a branch.

Instead, let's learn how to force this push operation to rewrite the history on this remote branch.

Problem

You wish to rewrite the history on the remote origin/inspiration branch based on the contents of the local inspiration branch.

Solution

- 1 Change to the directory containing your repository: for example, `cd /Users/mike/GitInPracticeRedux/`.
- 2 Run `git checkout inspiration`.
- 3 Run `git pull --rebase`.
- 4 Run `git push origin +inspiration` again. The output should resemble the following.

Listing 6.15 Output: rewritten history push

```
# git push origin +inspiration
```

```
Counting objects: 1, done.
```

```
Writing objects: 100% (1/1), 204 bytes | 0 bytes/s, done.
```

```
Total 1 (delta 0), reused 0 (delta 0)
```

```
To https://github.com/GitInPractice/GitInPracticeRedux.git
```

```
+ 0109344...ca74d2b inspiration -> inspiration (forced update)
```

1 Forced update

1 shows the git push output as usual but with a (forced update) indicating that it was forced to allow non-fast-forwards on the remote.

You've rewritten the history on a remote branch.

Discussion

You can also use `git push --force` instead of specifying the remote branch name prefixed with `+`, but this is not advised because it's less safe; depending on your Git configuration, you could accidentally force-push multiple branches at once. By default in some Git versions (which I tell you how to change later in section 7.1.2), a push will push all branches with matching local and remote branch names, so these will all be force-pushed if `git push --force` is run without parameters.

Remember that the reflog isn't pushed remotely, so if you unintentionally rewrite history on the remote branch, there's no way to recover commits you didn't have locally without direct access to the Git repository on the server. For this reason, you should be careful when rewriting remote branches. A good rule of thumb is to only ever do it on branches that nobody else is using. Avoid doing it on shared branches, and never do it on the master branch. Also, do a `git pull` immediately before any forced push to try to ensure that you aren't rewriting commits on the remote branch that you don't have locally. Of course, it's still possible for someone to push a commit just after you do a `git pull`, which you overwrite (which is why `git push --force` is a dangerous operation).

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

There are times when rewriting a few commits isn't enough; you want to rewrite the entire history of a branch. Perhaps you accidentally committed confidential files early in the project that you want to remove, or you want to split a large repository into multiple smaller ones.

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

To avoiding messing up your current local repository, let's create another cloned, local repository:

- 1 Change to the directory where you want the new `GitInPracticeRedux` repository to be created: for example, `cd /Users/mike/` to create the new local repository in `/Users/mike/GitInPracticeReduxFilterTest`.
- 2 Run `git clone https://github.com/GitInPractice/GitInPracticeRedux.git` `GitInPracticeReduxFilterTest` to clone into the `GitInPracticeReduxFilterTest` directory.
- 3 Change directory to the new Git repository: for example, `cd /Users/mike/GitInPracticeReduxFilterTest/`.

Figure 6.10 shows the partial output from `GitX` after the `git clone` command focusing on the master branch. Now let's remove references to the preface file from the master branch.

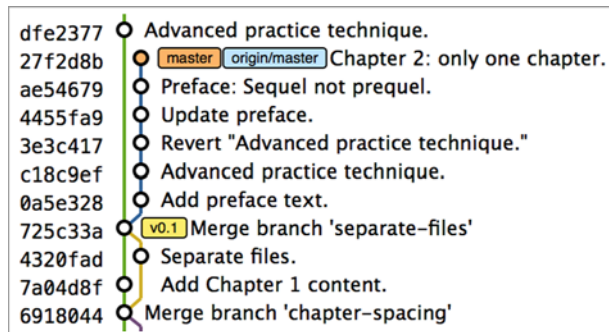


Figure 6.10 `GitX` before `filter-branch`

Problem

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

Solution

- 1 Change to the directory containing your filter test repository: for example, `cd /Users/mike/GitInPracticeReduxFilterTest/`.
- 2 Run `git filter-branch --prune-empty --index-filter "git rm --cached --ignore-unmatch 00-Preface.asciidoc" master`. The output should resemble the following.

Listing 6.16 Output: `filter-branch` file removal

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

Rewrite 4320fad6a58b105b8a1001f4f0da0258aa622feb (13/20)rm
'00-Preface.asciidoc'
Rewrite 725c33ace6cd7b281c2d3b342ca05562d3dc7335 (14/20)rm
'00-Preface.asciidoc'
Rewrite 0a5e3285e46900c7aa819d66e87d0c418a1c2f14 (15/20)rm
'00-Preface.asciidoc'
```

filter command 1 points to the `git filter-branch` command line.

Removed filename 4 points to the `'00-Preface.asciidoc'` strings in the rewritten commit messages.

2 remove command points to the `git rm --cached --ignore-unmatch 00-Preface.asciidoc` part of the command line.

3 Rewritten commit points to the `Rewrite` lines in the output.

```

Rewrite c18c9ef9adc73cc1da7238ad97ffb50758482e91 (16/20) rm
'00-Preface.asciidoc'
Rewrite 3e3c417e90b5eb3c04962618b238668d1a5dc5ab (17/20) rm
'00-Preface.asciidoc'
Rewrite 4455fa9c237f43e6b08f6190384579aa6ddad5cb (18/20) rm
'00-Preface.asciidoc'
Rewrite ae54679129ba8521265a750fc0e109add45414ac (19/20) rm
'00-Preface.asciidoc'
Rewrite 27f2d8b0a72427caf290e5127ab79533a0bc2867 (20/20) rm
'00-Preface.asciidoc'

```

Ref 'refs/heads/master' was rewritten  5 **Branch rewritten**

1 (git filter-branch) takes the following flags:

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

2 is passed as a string (surrounded by ") and is the command that's run by git filter-branch on each commit. Here git rm takes the following flags:

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

3 shows the first commit in which the index filter found the specified file (00-Preface.asciidoc). The newly created commit will have a different SHA-1 than the original commit listed here (4320fad6a58b105b8a1001f4f0da0258aa622feb).

4 shows the filename that was specified to git rm and that has been removed from this commit.

5 shows that the master branch was changed by the git filter-branch operation. If it wasn't, instead this would show `WARNING: Ref 'refs/heads/master' is unchanged`.

Figure 6.11 shows the output from GitX after the `git filter-branch` command from the first point at which they diverge (which is the first commit containing the 00-Preface.asciidoc file). All commits that referenced the file have been changed, and all those that only changed this file have been pruned, because they were empty. You can see that the origin/master remote branch also has an identical ref named refs/original/refs/heads/master. This is used as a backup so the original master can be restored by `git branch --force master refs/original/refs/heads/master` if desired.

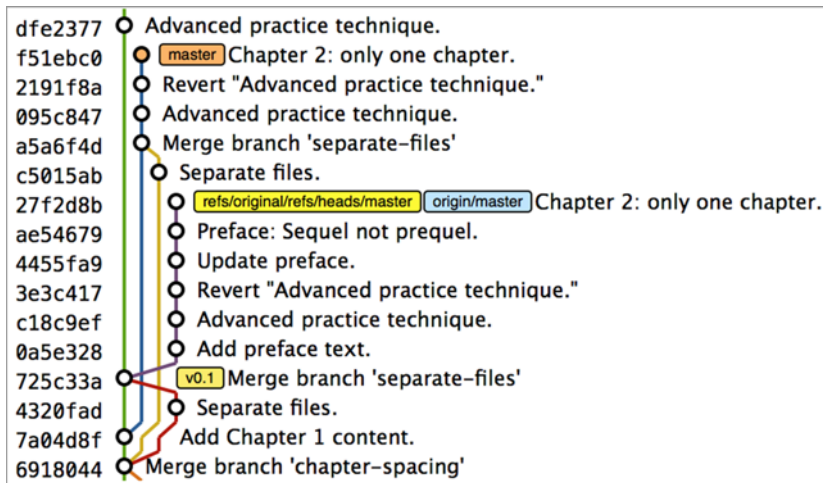


Figure 6.11 GitX after `filter-branch`

You have removed all references to `00-Preface.asciidoc` on the master branch.

Discussion

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

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

```
Cannot create a new backup.
A previous backup already exists in refs/original/
Force overwriting the backup with -f
```

- `--env-filter` lets you change environment variables to change the metadata for each commit. For example, you could set `GIT_AUTHOR_EMAIL` for each commit to change the email for every commit or change it conditionally to change it for a particular author.
- `--tree-filter` lets you rewrite the contents of the working directory tree. It checks out every revision and then allows modification of it. The command in this example could instead be done with a tree filter (`git filter-branch --tree-filter "rm 00-Preface.asciidoc" master`), but that would check out and delete the files from disk every time, which would be slower than the `--index-filter` used here.

- `--msg-filter` lets you rewrite commit messages. This can be useful in removing confidential information (or swear words) from commit messages.
- `--subdirectory-filter` filters the history to those commits that touch a particular subdirectory. It also makes that subdirectory the new project root. This can be useful when splitting a large repository into multiple smaller repositories based on existing subdirectories.
- `--parent-filter` changes commit parents. `--commit-filter` can vary commit commands, and `--tag-name-filter` modifies tag names, but these tend to be less widely used.

This book doesn't seek to cover all these flags in the same detail as this example. The best approach to learn more about this functionality is to read `git filter-branch --help` and experiment on duplicated local repositories.

6.2 Summary

In this chapter you learned the following:

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

Part 3

Advanced Git

Part 3 (chapters 7–11) provides some bonus Git skills that will help you be more efficient. These chapters introduce a collection of techniques that won't be used with every Git project but may be important for new ones.

This part will cover the following topics:

- How to set useful Git configuration settings and aliases
- How to show the current branch in a terminal prompt
- How to use submodules to vendor dependencies for a project
- How to send changes to and receive changes from a Subversion repository
- How to access a GitHub repository using a Subversion client
- How to create and merge GitHub pull requests and forks from the command line
- How to share a repository as a server or through a web interface



Personalizing Git

This chapter covers

- Setting the configuration for a single repository, all of a user's repositories, or all of a system's repositories
- Enabling useful configuration settings
- Aliasing, shortening, and chaining Git commands
- Using Git share configuration files between machines
- Showing the current branch in a terminal prompt

In this chapter, you'll learn about Git shortcuts. Git is a heavily configurable tool. As you've seen previously in this book, there are often times where you have multiple behaviors or choices for commands that you can select with flags or by using different commands. You may always want to run a command with a lengthy set of commands, or want to set your preferred `difftool` or `mergetool` as a default. These tweaks can be done per repository or globally, and they allow you to maximize productivity when adopting a particular Git workflow.

Technique 48 *Setting the configuration for all repositories*

In section 1.2 and technique 35, you saw how to set some configuration options using `git config`. You set the user and email address for the initial Git configuration and enabled `git rerere` to avoid having to resolve the same conflicts multiple times.

Problem

You wish to set your Git username in your global Git configuration.

Solution

- 1 You don't need to change to the directory of a particular or any Git repository.
- 2 Run `git config --global user.name "Mike McQuaid"`. There will be no output.

This sets the following value in my home directory: `/Users/mike/.gitconfig` (see the note “Where is the `$HOME` directory?”). You can read values from the configuration file by omitting the value argument. For example:

```
# git config --global user.name
Mike McQuaid
```

← Name value

You have set your Git username in your global Git configuration.

Discussion

When you run `git config --global user.name "Mike McQuaid"`, a file named `.gitconfig` is created (or modified if it exists) in your `$HOME` directory.

WHERE IS THE \$HOME DIRECTORY? The `$HOME` directory is often signified with the tilde (`~`) character, as it is in the rest of this chapter. If your username is `mike`, the `$HOME` directory typically resides in `C:\Users\mike` on Windows, `/Users/mike` on OS X, and `/home/mike` on Linux/Unix.

The filename is prefixed with a dot, and this means on OS X and Linux that it may be hidden by default in graphical file browsers or file dialogs. If you run `cat ~/.gitconfig` in a terminal, you can see the contents. Provided you ran `git config` as requested in section 1.2 and technique 35, it should look something like this:

```
# cat ~/.gitconfig
[user]
name = Mike McQuaid
email = mike@mikemcquaid.com
[rerere]
enabled = 1
```

← User section ← Email key and value

You can see that these commands create two sections (user and rerere) and three values (name, email, and enabled). The `git config` command takes arguments in the format `git config --global section.key value`. If you ran this command again with the same `section.key` but a different value, it would alter the current value rather than creating a new line.

This `~/.gitconfig` file is used to set your preferred configuration settings to be shared among all your repositories on your current machine. You could even share this file between machines to allow these settings to be used everywhere. This is detailed later, in section 7.2.

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

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

Now that you've seen how to set and read some configuration settings for all repositories, let's see how to do it for a single one.

Technique 49 *Setting the configuration for a single repository*

There are times when you may want to use different configuration settings for different repositories on the same computer. For example, in the past I've used one email address when committing to open source repositories and another email address when committing to my employer's repositories. If you wanted to do both of these on the same computer, you could set a different `user.email` value in the single repository configuration file to be used in preference to the global `~/.gitconfig`.

Recall that whenever you've used `git config` previously, you've always used the `--global` flag. But you can use four different flags to affect the location of the configuration file that's used:

- `--global`—Uses the `~/.gitconfig` file in your `$HOME` directory. For example, if your `$HOME` was `/Users/mike`, then the global file would be at `/Users/mike/.gitconfig`.
- `--system`—Uses the `etc/gitconfig` file under wherever Git was installed. For example, if Git was installed into `/usr/local/`, the system file would be at `/usr/local/etc/gitconfig`; or if installed into `/usr/`, the system file would be at `/etc/gitconfig`.
- `--local`—Uses the `.git/config` file in a Git repository. For example, if a Git repository was at `/Users/mike/GitInPracticeRedux/.git`, then the local file would be at `/Users/mike/GitInPracticeRedux/.git/config`. `.git/config` is the default if no other configuration location flags are provided.
- `--file` (or `-f`)—Takes another argument to specify a file path to write to. For example, you could specify a file using `git config --file /Users/mike/Documents/git.cfg`.

Problem

You wish to set your Git user email in your repository Git configuration.

Solution

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

- 2 Run `git config user.email mike.mcquaid@github.com`. There will be no output.

The email address doesn't need to be surrounded with quotes because it has no spaces, unlike a name such as "Mike McQuaid".

This sets the value in the `.git/config` file in the repository. You can query it as follows:

```
# git config --local user.email
mike.mcquaid@github.com
```

← **user.email value**

You have set your Git user email in your repository Git configuration.

Discussion

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

- 1 The argument following `--file` (if provided)
- 2 The local configuration file (`.git/config`)
- 3 The global configuration file (`~/.gitconfig`)
- 4 The system configuration file (etc/`gitconfig` under where Git was installed)

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

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

Listing 7.1 Sample `.git/config` file

```
# cat .git/config

[core]
  repositoryformatversion = 0
  filemode = true
  bare = false
  logallrefupdates = true
  ignorecase = true
  precomposeunicode = false
[remote "origin"]
  url = https://github.com/GitInPractice/GitInPracticeRedux.git
  fetch = +refs/heads/*:refs/remotes/origin/*
[branch "master"]
  remote = origin
  merge = refs/heads/master
[branch "inspiration"]
  remote = origin
  merge = refs/heads/inspiration
[user]
  email = mike.mcquaid@github.com
```

You can see that various default options are set based on the current system (such as `ignorecase`, because Git has detected that I'm using the default OS X case-insensitive filesystem) and interactions with the Git repository. When you do a `git push --set-upstream`, Git sets values in a branch section in the `.git/config` file. This section specifies where to push and pull from when on a certain branch.

7.1 Useful configuration settings

In this section, I'll show you how to set some of the most useful configuration settings that make Git easier to use. But Git has a huge number of configuration settings, and it would take a significant proportion of this book to detail them all. I recommend reading `git config --help` at some point and considering which other settings you may wish to change. Additionally, in appendix C, you can see my personal, commented Git configuration if you're interested in what I use.

7.1.1 Colored output in Git

Colored output was enabled by default in Git 1.8.4. As a result, if your installed version of Git is 1.8.4 or above (check by running `git --version`), you can skip this section.

Git's output doesn't use colors by default on versions below 1.8.4. To enable colored Git output, you can run the following:

```
# git config --global color.ui auto
```

This means that, if supported by your terminal, and if Git isn't writing the output of a command to a file, Git will use colored text in the output. I think colored output makes Git's commands much easier to read and parse quickly. The `git diff` output in this case uses red for removed lines and green for added ones. This is a much quicker way of parsing these changes than looking for a `+` or `-` symbol (which is included in the output regardless).

Note that the red and green colors chosen are set by your terminal rather than Git. If you wish to change them, you'll need to change your terminal's settings (which are specific to the terminal software you're using).

7.1.2 Git 2.0's push defaults

Git 2.0 (which was released on May 28, 2014) defaulted to a new push strategy: the simple push strategy. This means branches are pushed to their upstream branch (set the first time with `git push --set-upstream`). Also, with the simple strategy, Git refuses to push if the remote branch name is different than the local branch name unless you specify it with an option such as `git push origin remotibranchname`. Because this is the new behavior, it's a good idea to enable it in older versions of Git. If your installed version of Git is 2.0 or above (check by running `git --version`), you can skip this section.

Git versions below 2.0 use the matching strategy for their default push behavior. This means when you run `git push` without arguments, Git will push all branches that

have the same local and remote branch name. For example, if you have master and inspiration local branches and origin/master and origin/inspiration remote branches, then when you run `git push`, any changes made on both master and inspiration local branches will be pushed to their remote branches. I think this is confusing; when on a branch, I would expect `git push` to only affect the branch I'm on. Let's switch to the simple strategy instead by running the following:

```
# git config --global push.default simple
```

I always enable this if I have to use older Git versions, and I highly recommend you do, too; it means you're less likely to accidentally push changes made on other branches that aren't ready to be pushed yet.

7.1.3 **Pruning branches automatically**

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

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

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

```
# git config --global fetch.prune 1
```

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

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

You've already seen in technique 21 how you can use a `.gitignore` file to ignore certain files within a repository. Sometimes you may have problems with this approach; other users of the repository may disagree about what files should be ignored, or you may be sick of ignoring the same temporary files your editor generates in every repository you

use. For this reason, Git allows you to set a global ignore file in which you can put your personal ignore rules (useful if others don't want them in a repository). To tell Git you wish to use a ~/.gitignore file, run the following:

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

This global file behaves like any other .gitignore file, but you can put entries in it to be shared among all repositories. For example, in mine I put .DS_Store, which are the thumbnail cache files that OS X puts in any directory you view with Finder.app that contains images (you can see it in appendix C). I also put in it editor-specific files and build output directory names that I tend to personally prefer. This means I don't need to remember to do so for every new repository I use or add an ignore rule to repositories whenever I change text editors.

7.1.5 **Displaying help output in your web browser**

You might be someone who keeps their web browser open more than a terminal or finds documentation easier to read in a browser than a terminal. You can request that `git --help` commands display their output in a web browser by appending the `--web` flag. For example, to get help for the `git help` command in the web browser, you run `git help --help --web`.

This may fail with the message `fatal: HTML documentation is not provided by this distribution of git`. This is because some Git installations don't install HTML documentation. If this is the case, you can find the Git HTML documentation at <http://git-scm.com/docs/> and skip the rest of this section.

If your Git installation displays the HTML documentation correctly, you can tell `git help` and `git --help` to always display documentation in HTML format by running the following:

```
# git config --global help.format web
```

After this, when you run a command like `git config --help`, instead of displaying in your terminal, it will open the HTML documentation in your default browser. If you wish to configure the browser that's used, you can run `git web--browse --help` to view the many different ways of doing so.

7.1.6 **Storing passwords in the OS X keychain**

Apple's Mac OS X operating system provides a system-wide secure keychain for each user. This is what's used to store your passwords for various services such as network shares. You can also request that Git store its various passwords there: for example, for private [https:// GitHub](https://github.com) repository clones. To do this, run the following:

```
# git config --global credential.helper osxkeychain
```

After setting this, the next time you clone a private GitHub repository and ask for a password, you'll be prompted whether to allow `git-credential-osxkeychain` access to your keychain. You should allow this, and then passwords will be stored and

retrieved from here in future. This is useful on OS X, because otherwise Git may prompt for the same passwords multiple times or write them unencrypted to disk.

Alternatively, on Windows, there's a tool named `git-credential-winstore` (available at <http://gitcredentialstore.codeplex.com>) to store these credentials in the Windows Credential Store. On Linux/Unix, there's a tool named `git-credential-gnome-keyring` (bundled with Git 1.8.0 and above) to store these credentials in the GNOME Keyring.

7.1.7 **Storing arbitrary text in Git configuration**

In addition to all the supported keys, you can use any Git configuration file as an arbitrary key-value store. For example, if you ran `git config --global gitinpractice.status inprogress`, these lines would be added to your `~/.gitconfig` file:

```
# git config --global book.gitinpractice.firstedition.status inprogress
[book "gitinpractice.firstedition"]
    status = inprogress
```

These could then be retrieved using `git config book.gitinpractice.firstedition.status`. Git silently ignores any configuration values it doesn't recognize. This allows you to use the Git configuration file to store other useful data. I use it to store configuration data for some personal shell scripts. For example, I store my SourceForge username in `sourceforge.username` so scripts unrelated to Git can run `git config sourceforge.username` to get the username.

7.1.8 **Autocorrecting misspelled commands**

If you often mistype commands—such as `git pish` instead of `git push`—you could set up an alias. But it may be time-consuming and clutter up your configuration file to do this for every variant you mistype. Instead, you can enable Git's autocorrection feature by running the following:

```
# git config --global help.autocorrect 1
```

This waits for the value-specified number of 0.1 seconds (a value of 2 would wait 0.2 seconds) before autocorrecting and running the correct version. You may wish to set this time to longer if you wish to verify the command before it runs.

For example, here's the result if I run `git pish` after this configuration change:

```
# git pish
WARNING: You called a Git command named 'pish', which does not exist.
Continuing under the assumption that you meant 'push'
in 0.1 seconds automatically...
Everything up-to-date
```

If the wrong command is going to be run, you can press Ctrl-C to cancel it after the WARNING text is displayed.

Technique 50 *Aliasing commands*

One of the most powerful features available with `git config` is aliasing. Aliases allow you to create your own Git commands from combinations of other Git commands or by renaming them. This may be useful for making commands that are more memorable or easier to type. These are set as configuration values in the `alias` section.

Problem

You wish to create a shorter alias for the ultimate log output from section 4.4.

Solution

- 1 You don't need to change to the directory of a particular or any Git repository.
- 2 Run `git config --global alias.ultimate-log "log --graph --oneline --decorate"`. There will be no output.

You can verify that this has worked by viewing the relevant section of the `~/.gitconfig` file using `grep`.

Listing 7.2 Output: ultimate log alias

```
# grep --before=1 ultimate ~/.gitconfig
[alias]
  ultimate-log = log --graph --oneline --decorate
```

← Alias section
← Alias value

You've created an alias named `ultimate-log`. Now if you run `git ultimate-log`, it's equivalent to running `git log --graph --oneline --decorate`. Any arguments that follow `git ultimate-log` will be treated the same as arguments following `git log --graph --oneline --decorate`.

Discussion

It's easier to remember `ultimate-log` than the various flags, but it's still unwieldy to type. If you use `git ultimate-log` all the time, you may want to have fewer characters to type. Aliases can be any length, so you can make a shorter value using `git config --global alias.l '!git ultimate-log'`:

```
# git config --global alias.l '!git ultimate-log'
"log --graph --oneline --decorate"
# grep --before=1 ultimate ~/.gitconfig
[alias]
  ultimate-log = log --graph --oneline --decorate
  l = !git ultimate-log
```

Note the use of single quotes when setting the alias. These are required in this case: otherwise the Unix shell might not write the `!`, and you'd see an error like: `Expansion of alias 'l' failed; 'ultimate-log' is not a git command.`

Now you can use `git l` to run `git ultimate-log`, which will in turn run `git log --graph --oneline --decorate`. You may wonder why you don't just set `git l` to be the ultimate log directly, rather than passing through another command. I prefer to do this as a way of making the `.gitconfig` file easier to read and follow.

In addition to adding a longer version of the command, you may want to add comments into your Git configuration files. You can do this by manually prefixing any line with the `#` or `;` character. For example, in my `~/.gitconfig` I have the following:

```
[alias]
## 'New' Commands
# Show the commit log with a prettier, clearer history.
pretty-one-line-log = log --graph --oneline --decorate

## Shortened 'New' Commands
l = !git pretty-one-line-log
```

Using this format of comments, longer commands, and shortened ones helps make your `.gitconfig` file easier to follow. When you or someone else looks back on the changes you made, the comments and more verbose commands make it more obvious what your reasons were for adding each section.

In addition to aliasing and shortening commands, you can also use the alias functionality to chain multiple commands together. Any alias that starts with a `!` is run as a command in the root of the repository's working directory. Let's create a command that does a fetch and then an interactive rebase.

Run `git config --global alias.fetch-and-rebase '!git fetch && git rebase -i origin/master'`. This tells Git to go to the root of the working directory (the directory containing the `.git` directory), run `git fetch`, and, if it succeeds, run `git rebase -i origin/master`.

This can be useful in doing something similar to `git pull --rebase` but doing an interactive rebase instead. I often use this when I know some changes have been made upstream and I want to squash and reorder my commits based on these changes. For example, if I know changes have been made to the `origin/master` remote branch, this alias will fetch them and interactive-rebase the current branch on top of the `origin/master` remote branch so I can do the various things described in technique 44.

7.2 **Sharing your Git (or other) configuration between machines**

Some people use Git on multiple machines. You may use it on both a desktop and a laptop computer. It's annoying to have your configuration be different on each machine, so you may wish to keep your `~/.gitconfig` settings in sync so they're the same on every machine.

A common solution is to create a *dotfiles* repository on GitHub. This involves creating a Git repository; adding all your Git global configuration files such as `~/.gitconfig` and `~/.gitignore`; and committing, pushing, and sharing these files between machines as you would any other Git repository. This can be good practice for learning how to

use Git. You can use dotfiles repositories to share many other application configuration files (such as a `.bashrc` file to configure the Bash shell).

You may be interested in my dotfiles repository on GitHub (<https://github.com/mikemcquaid/dotfiles>). It contains various configuration files including my `.gitconfig` and `.gitignore`, which are well documented (and included in this book in appendix C). I've also created a simple script named `install-dotfiles.sh`. After cloning my dotfiles repository to somewhere in my `$HOME` directory, I can run `install-dotfiles.sh` to symlink or copy all the dotfiles files into their correct locations. This means I can easily get and install all my dotfiles on any machine that has Git installed. This is useful because I use the same dotfiles across my multiple computers, virtual machines, and servers.

GitHub also provides a dotfiles page with some notable dotfiles repositories and discussion of why they're useful at <http://dotfiles.github.io>.

Technique 51 *Showing the current branch in your terminal prompt*

As you've noticed throughout this book, it's common to create and change branches frequently when using Git. When using multiple repositories or not using one for a while, it may be difficult to remember what branch is currently checked out. You could run `git branch`, but if you're switching regularly among multiple repositories, it can be handy to have this information displayed in your terminal. Let's learn how to do this for two popular shells: Bash and ZSH.

Problem

You wish to add the current Git branch to your Bash or ZSH terminal prompt.

Solution

First, determine what shell you're using by running `basename $SHELL`. This should output either `bash` or `zsh`. If it outputs something else, you may need to modify the instructions (which, I'm afraid, is beyond the scope of this book).

Add the following function to your `~/.bashrc` file if you're using Bash or `~/.zshrc` file if you're using ZSH:

```
git_branch() {  
  GIT_BRANCH=$(git symbolic-ref --short HEAD 2>/dev/null) || return  
  [ -n "$GIT_BRANCH" ] && echo "($GIT_BRANCH) "  
}
```

This provides a `git_branch` function. Once you've added it, open a new shell, `cd` to a Git repository, and run `git_branch`. If you're on the master branch, the output should be `(master)`.

This function uses the `git symbolic-ref` command, which resolves a ref to a branch. In this case you're asking for the shortest branch ref for the `HEAD` pointer—the currently checked-out branch. This is then output surrounded with brackets.

Let's make a prompt of the format `hostname (branch) #`. If you're using Bash, add the following to `~/.bashrc`:

```
PS1='\[\033[01;32m\]\h \033[01;31m\}${git_branch}\
\[\033[01;34m\]#\[\033[00m\] '
```

If you're using ZSH, add the following to `~/.zshrc`:

```
autoload -U colors && colors
PROMPT='%{$fg_bold[green]}%m %{$fg_bold[red]}${git_branch}\
%{$fg_bold[blue]}%# %b%f'
```

The differences between the two reflect the different ways of setting colors in Bash and ZSH and the different variables that are used to output the hostname (`\h` versus `%m`) and the colors (`\[\033[01;32m\]` versus `%{$fg_bold[green]}%`).

Be careful to enter them exactly as is, or they may cause errors. You may wish to enter them into your currently running terminal to test them before inserting into your `~/.bashrc` or `~/.zshrc`. The final version should look something like figure 7.1.



Figure 7.1 Shell branch output

You have successfully added the current Git branch to your Bash or ZSH terminal prompt.

Discussion

This prompt works by running the `git symbolic-ref --short HEAD` command every time a new prompt is displayed. In event of an error or no output (no checked out branch), it won't display any Git information in the prompt.

7.3 Summary

In this chapter you learned the following:

- How to use `git config` to set and get values from `.git/config`, `~/.gitconfig`, and `etc/gitconfig`
- How to set various useful values from those listed by `git config --help`
- How to create a `git ultimate-log` command and shorten it to `git l`
- How to create a `git fetch-and-rebase` command that runs `git fetch` and then `git rebase --interactive`
- How to use a dotfiles repository to share configuration files between machines
- How to make a Bash or ZSH terminal prompt use the `hostname (branch) #` format

Vendoring dependencies as submodules

This chapter covers

- When submodules are useful
- Adding a submodule to a repository
- Viewing the status of the submodules in a repository
- Updating and initializing all submodules in a repository
- Running a command in every submodule in a repository

In this chapter, you'll learn how to maintain dependencies on other Git-based software projects within your own using Git submodules.

8.1 *When are submodules useful?*

Almost all software projects use other software projects as libraries or tools. For example, say you're using Git and writing a desktop application in C++, and you want to communicate with a server that provides a JSON API. Rather than writing the JSON-handling code yourself, you find an open source project on GitHub that provides a library for accessing JSON APIs with C++. You want to include this open source library into your project and update it when new versions are released with bug fixes you need.

There are generally two approaches to handling other software projects (usually known as *dependencies*) and which versions work with your own software:

- Write documentation for what other software projects are required, what versions they should be, and where they should be installed, so other developers building the project know how to set it up correctly.
- Include the dependencies in the project's repository so they're always available to anyone when cloning the repository. This is known as *vendoring* the dependencies.

Both approaches have pros and cons. Adopting the first means the software source code repository can avoid including other software projects. In the C++ application example, it would mean documenting for other developers where and how they should download the external JSON library's Git repository rather than storing anything related to it in the C++ application's Git repository.

Using the second approach means you always have the various dependencies available but can increase the space used by the version control system. In the C++ application example (without submodules), you might copy the source code of the external JSON library into the application's Git repository. When you wanted to update the library version, you'd copy in the new code and commit it.

Because Git stores the complete history of a repository and downloads it all when cloned, too many large dependencies can result in a repository that takes a long time to clone and is unclear about any other repositories whose source code was used to populate some of this repository. This makes updating versions of things like external libraries a painful, manual process. For this reason, submodules were created.

A Git repository can contain *submodules*. They allow you to reference other Git repositories at specific revisions. This technique is most commonly used to reference external Git repositories that are dependencies for software projects. In the C++ application example, instead of documenting the location or copying the source code of the external JSON library into the application's Git repository, you could use submodules to reference the external JSON library's Git repository.

Git's submodules store the reference to a specific SHA-1 reference in another remote Git repository and store these in subdirectories of the current repository. All that is actually committed in the current repository are some small pieces of metadata that the `git submodule` command uses to clone, update, and interact with the various submodules in a Git repository.

WHAT IS GIT SUBTREE? You may have heard about `git subtree`, which is an alternate method of managing Git subprojects in a Git repository. Instead of just referencing other Git repositories, `git subtree` stores the contents of the remote Git repository. It's a contributed command to Git, which means it's not documented or supported to the same extent as `git submodule`, so I won't be covering it in this book. If you want to read more, you can view the `git subtree` documentation on GitHub: <https://github.com/git/git/blob/master/contrib/subtree/git-subtree.txt>.

Technique 52 Adding a git submodule: git submodule add

Let's start by creating a new repository that can be used as a submodule in the existing GitInPracticeRedux repository.

Create a new repository on GitHub that you can use as a submodule by following these steps:

- 1 Create a new repository with `git init`, and pass the path to a new directory: for example, `git init /Users/mike/GitInPracticeReduxSubmodule/`.
- 2 Change to the directory containing your new submodule repository: in this example, `cd /Users/mike/GitInPracticeReduxSubmodule/`.
- 3 Create a file named `TODO.md` with the `echo` command by running `echo "# TODO\n1. Add something useful to this submodule." > TODO.md`.
- 4 Commit the new `TODO.md` file as the initial commit by running `git commit --message "Initial commit of submodule." TODO.md`.
- 5 Create a new repository on GitHub (or another Git hosting provider).
- 6 Add the new remote reference to the GitHub repository by running `git remote add origin https://github.com/GitInPractice/GitInPractice-ReduxSubmodule.git`.
- 7 Push the repository to GitHub by running `git push --set-upstream origin master`.

The output of all these commands should resemble the following.

Listing 8.1 Output: creating a new submodule repository

```
# git init /Users/mike/GitInPracticeReduxSubmodule/
Initialized empty Git repository in
/Users/mike/GitInPracticeReduxSubmodule/.git/
# cd /Users/mike/GitInPracticeReduxSubmodule/
# echo "# TODO\n1. Add something useful to this submodule." > TODO.md
# git commit --message "Initial commit of submodule."
[master (root-commit) e95b4cd] Initial commit of submodule.
1 file changed, 2 insertions(+)
create mode 100644 TODO.md
# git remote add origin
https://github.com/GitInPractice/GitInPracticeReduxSubmodule.git
# git push --set-upstream origin master
Counting objects: 3, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 272 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/GitInPractice/GitInPracticeReduxSubmodule.git
 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.
```

← 1 New repository

← 2 Initial commit

← 3 Push repository

- ❶ shows the creation of a new Git repository on disk to be used as a new submodule repository for the `GitInPracticeRedux` repository. It has been created outside the `GitInPracticeRedux` directory, so it can be added later as if it were just another GitHub repository.
- ❷ shows the first commit to the new submodule repository of the `TODO.md` file.
- ❸ shows the push of the initial commit to the newly created GitHub repository.

The new submodule repository has been created and pushed to GitHub. Note that it's not yet a submodule of the `GitInPracticeRedux` repository; this was just to create a new repository that could be added as a submodule repository afterward.

Now that the submodule repository has been created and pushed to GitHub, it can be removed from your local machine with `rm -rf GitInPracticeReduxSubmodule/`. Don't worry; a complete copy is stored on GitHub (which you'll use next).

Now that you've created a new submodule repository, let's add it as a submodule to the existing repository.

Problem

You wish to add a the `GitInPracticeReduxSubmodule` repository as a submodule of the `GitInPracticeRedux` repository in 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 submodule add https://github.com/GitInPractice/GitInPractice-ReduxSubmodule.git submodule`.
- 4 Commit the new submodule changes to the repository by running `git commit --message "Add submodule."`.

The output of all these commands should resemble the following.

Listing 8.2 Output: adding a submodule

```
# git submodule add
https://github.com/GitInPractice/GitInPracticeReduxSubmodule.git
submodule

Cloning into 'submodule'...
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 3 (delta 0)
Unpacking objects: 100% (3/3), done.
Checking connectivity... done.

# git commit --message "Add submodule."

[master cc206b5] Add submodule.
2 files changed, 4 insertions(+)
```

←❶ **Submodule clone**

```
create mode 100644 .gitmodules
create mode 160000 submodule
```



- ❶ shows the clone of `GitInPracticeReduxSubmodule` into the directory named `submodule` in the local repository. After this was done, it also created a `.gitmodules` file in the root of the repository's working directory.
- ❷ shows the file that contains the submodule metadata, such as the directory path and the URL.
- ❸ shows the new directory named `submodule` that was created to store the contents of the new submodule repository. Note that you normally wouldn't use this directory name—you're just using it for these examples.

You have successfully added the `GitInPracticeReduxSubmodule` submodule to the `GitInPracticeRedux` repository.

Discussion

The new directory named `submodule` behaves like any other Git repository. If you change into its directory, you can run services like `GitX` and `git log`, and even make changes and push them to the `GitInPracticeReduxSubmodule` repository (provided you have commit access).

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

Initializing all submodules can be done by running `git submodule init`, which copies all the submodule names and URLs from the `.gitmodules` file to the local repository Git configuration file (in `.git/config`). Note that this was done for you when you ran `git add`.

Let's take a closer look at the last commit.

Listing 8.3 Output: `git show submodule`

```
# git show
commit cc206b5c9b30eef23578e48dadfa3b194a50cfe7
Author: Mike McQuaid <mike@mikemcquaid.com>
Date:   Fri Apr 18 16:16:30 2014 +0100
```

```
    Add submodule.
```

```
diff --git a/.gitmodules b/.gitmodules
new file mode 100644
index 0000000..c63f995
--- /dev/null
+++ b/.gitmodules
@@ -0,0 +1,3 @@
+[submodule "submodule"]
+    path = submodule
```




```

+      url = https://github.com/GitInPractice/GitInPracticeReduxS...
diff --git a/submodule b/submodule
new file mode 160000
index 0000000..e95b4cd
--- /dev/null
+++ b/submodule
@@ -0,0 +1 @@
+Subproject commit e95b4cd02cafa486a7baec19ab26edec28e9eddc

```

Submodule URL ③

④ Submodule commit

- ① shows the name of the submodule that was created in the repository: `submodule`. This is used to reference this particular submodule with any additional submodule commands.
- ② shows the directory location where the submodule is cloned. This is where the submodule files will be accessed.
- ③ shows the remote repository location for the submodule that was added.
- ④ shows the commit SHA-1 for the submodule. Even if there are changes to the submodule, this will always be the commit that is checked out by anyone using this submodule in this repository. This is to ensure that the submodule only uses a known, tested version and that changes to the submodule's Git repository (which may be something you don't have any control over) don't change anything in the current repository.

`git submodule add` can also take some parameters to affect its behavior:

- `--quiet (-q)`—Makes `git submodule add` only print out error messages and no status information.
- `--force (-f)`—Lets you add a submodule path that would otherwise be ignored by `.gitignore` rules.
- `--depth`—Passed to the `git clone` of the submodule to let you create a shallow clone containing only the requested number of revisions. This can be used to shrink the size of the submodule on disk. This flag for `git clone` was mentioned previously in technique 8 and can be useful for reducing the clone time for very large repositories.

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

Now that you've added a submodule to the repository, it can be useful to query what submodules have been added and what their current status is. This can be done with the `git submodule status` command.

Problem

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

Solution

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

Listing 8.4 Output: submodule status

```
# git submodule status
e95b4cd02cafa486a7baec19ab26edec28e9eddc submodule (heads/master)
```

Submodule status ①

- ① shows the SHA-1 of the pinned submodule, the name, and the ref it's pointing to (the `master` branch in this case). This matches the SHA-1 you saw earlier in the submodule directory metadata.

Discussion

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

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

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

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

- 1 Change to the directory containing your repository: for example,
`cd /Users/mike/GitInPracticeRedux/`.
- 2 Run `git submodule deinit`.
- 3 Run `rm -rf .git/modules/` to remove the directory where Git caches submodules outside of the working directory.

The output of all these commands should resemble the following.

Listing 8.5 Output: removing a submodule clone

```
# git submodule deinit .
Cleared directory 'submodule'
Submodule 'submodule'
  (https://github.com/GitInPractice/GitInPracticeReduxSubmodule.git)
  unregistered for path 'submodule'
# rm -rf .git/modules/
```

← ① Submodule deinit
 ← ② Submodule unregister
 ← ③ Submodule delete

- ❶ shows the clearing of the submodule directory. This means the directory named submodule has all its contents deleted.
- ❷ shows that the submodule has been unregistered—this doesn't remove the submodule from the repository but returns it to an uninitialized state.
- ❸ shows the deletion of the submodule storage directory. Although the submodule is cloned into the submodule directory, it's initially cloned into `.git/modules` too and then cloned from there to the submodule directory. Deleting this ensures that there's no copy of the submodule's repository in the current repository.

Sometimes, at the same time as initializing a repository, you may want to update it to the latest revision to incorporate any changes made in the upstream, submodule repository. Now that you've removed the submodule from your repository, you can initialize the submodule and update it to any later revision.

Problem

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

Solution

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

Listing 8.6 Output: initializing and updating a submodule

```
# git submodule update --init

Submodule 'submodule'
  (https://github.com/GitInPractice/GitInPracticeReduxSubmodule.git)
  registered for path 'submodule'
Cloning into 'submodule'...
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 3 (delta 0)
Unpacking objects: 100% (3/3), done.
Checking connectivity... done.
Submodule path 'submodule': checked out
  'e95b4cd02cafa486a7baec19ab26edec28e9eddc'
```

❶ Submodule init

❷ Submodule clone

❸ Submodule checkout

- ❶ shows the registration of the submodule into the Git repository.
- ❷ shows the submodule being cloned into the local Git repository.
- ❸ shows the submodule contents being checked out into the submodule directory for the currently stored revision.

Discussion

If there were any changes to the `GitInPracticeReduxSubmodule` repository, the `git submodule update --init` command would initialize the submodule in the local

repository and then update the stored submodule revision to the latest revision in the local repository. This would then require another commit and push to update the remote repository. This should only be done after testing that the changes made to the `GitInPracticeReduxSubmodule` repository remain compatible with the `GitInPracticeRedux` project.

`git submodule update` can take some parameters to customize its behavior:

- `--recursive`—Runs `git submodule update --init` in each of the submodules directories. This is useful when there are nested submodules inside submodules.
- `--no-fetch`—Attempts to update the submodule without running `git fetch`. This updates the submodule to a later revision only if it has already been fetched. This is useful if you want to fetch the changes to a submodule now and then update and test this update at a later point.
- `--force (-f)`—Updates the submodules to the latest revision by running the equivalent of `git checkout --force` to discard any uncommitted changes made to the submodules.
- `--depth`—Passed to the `git clone` of the submodule to let you create a shallow clone containing only the requested number of revisions. This can be used to shrink the size of the submodule on disk.

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

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

Sometimes you may wish to perform a command or query in every submodule. For example, you may want to iterate through all the submodules in a repository (and their submodules) and run a Git command to ensure that they have all checked out the master branch and have fetched the latest remote repository commits or print status information. Git provides the `git submodule foreach` command for this case: it takes a command (or commands) as an argument and then iterates through each Git submodule (and their submodules) and runs the same command.

Problem

You wish to output some status information for every submodule in the `GitInPracticeRedux` repository.

Solution

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

- 2 Run `git submodule foreach 'echo $name: $toplevel/$path [$sha1]'`. The output should resemble the following.

Listing 8.7 Output: submodule loop

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

Entering 'submodule'

```
submodule: /Users/mike/Documents/GitInPracticeRedux:submodule
[e95b4cd02cafa486a7baec19ab26edec28e9eddc]
```

1 Current submodule

2 Submodule name, path

3 Submodule SHA-1

- 1 shows a message with the name of each submodule that is iterated through.
- 2 shows the use of the `git submodule foreach` `$name`, `$toplevel`, and `$path` variables to print out the name of the submodule, the top-level repository it belongs to, and the path within that repository.
- 3 shows the use of the `git submodule foreach` `$sha1` variable to print the current SHA-1 of the submodule.

You have successfully iterated through the submodules in the `GitInPracticeRedux` repository and used all the `git submodule foreach` variables to print status information.

Discussion

`git submodule foreach` can take the following flags:

- `--quiet`—Only prints command output, not the “Entering *submodule*” message, as it runs on each submodule
- `--recursive`—Iterates through any submodules that exist for any of the submodules

8.2 Summary

In this chapter you learned the following:

- How to use submodules to vendor project dependencies
- How to use `git submodule add` to add a submodule and commit its metadata
- How to use `git submodule status` to view all submodules and their current revision
- How to use `git submodule update --init` to initialize all submodules, fetch any changes, and update them to the latest revision
- How to use `git submodule foreach` and its variables to run commands and print metadata for every submodule in a repository

Working with Subversion



This chapter covers

- Importing an existing Subversion repository into a Git repository
- Fetching from and pushing changes to a Subversion repository for Git
- Accessing a GitHub repository using a Subversion client

Despite the growth of Git, Subversion (also known as SVN) remains widely used across many projects. Many older open source projects continue to use Subversion, as do many internal corporate projects.

Fortunately, Git provides functionality to allow you to access existing Subversion repositories through the `git svn` command. This also provides a good way of learning Git if you have to use Subversion repositories, and a good migration path for projects that currently use Subversion and wish to use Git. In this chapter you'll learn about Git's Subversion integration.

Technique 56 Importing an SVN repository into a Git repository

Let's start by importing the "Google Search Appliance connector for Lotus Notes" Subversion repository from Google Code. I selected it because it has a small history but contains branches and tags so you can see how they're used.

HOW CAN YOU INSTALL GIT-SVN? `git svn` is usually installed as part of the default Git installation (and is in all the official Git installers). If it hasn't been installed, then when you run `git svn`, you'll see a message resembling this: `WARNING: You called a Git command named 'svn', which does not exist. In this case you'll need to install git-svn separately. You can do so with your package manager: for example, on Debian/Ubuntu run apt-get install git-svn.`

Problem

You wish to import a Subversion repository into a local Git repository.

Solution

- 1 Change to the directory where you want the new google-notes repository to be created: for example, `cd /Users/mike/` to create the new local repository in `/Users/mike/GitInPracticeRedux`.
- 2 Run `git svn clone --prefix=origin/ --stdlayout http://google-enterprise-connector-notes.googlecode.com/svn/ google-notes`. The output should resemble the following.

Listing 9.1 Output: cloning a Subversion repository

```
# git svn clone --prefix=origin/ --stdlayout
http://google-enterprise-connector-notes.googlecode.com/svn/
google-notes
```

1 clone command

2 SVN repository

3 Directory name

```
Initialized empty Git repository in /Users/mike/google-notes/.git/
r1 = ec0d880208029f06e2181ce2241d5f950b77f8c2
(refs/remotes/origin/trunk)
A    domino_stylesheets.xslt
r2 = ad41618c83d7dfd2768c8668bfdce0a63ea6cfca
(refs/remotes/origin/trunk)
A    google-access-control-1_0.ntf
...
r381 = 750dd5077c4122bcc3a8a17a65088c0ebd85a2b6
(refs/remotes/origin/trunk)
Checked out HEAD:
http://google-enterprise-connector-notes.googlecode.com/svn/trunk
r381
```

4 First revision

5 Added file

6 Last revision

7 HEAD position

- 1 shows the `git svn clone` command used to clone the entire Subversion repository. The `--prefix` argument is only needed in Git versions before 2.0 (which was released on May 28, 2014) and means the Subversion remote branches will all be created under `origin/` (like a normal Git remote repository). The `--stdlayout` flag is used to say that the Subversion repository is in the usual format—it has

branches, tags, and trunk as the subfolders of the root of the repository, and they should all be imported.

- ❷ is the HTTP URL of the Subversion repository you've cloned.
- ❸ is the name of the local directory that will contain the new Git repository created from the Subversion repository.
- ❹ shows the first commit/revision (r1) of the Subversion repository being mapped and imported as a new commit/SHA-1 (ec0d88...) into the local Git repository.
- ❺ shows a new file added in the first imported commit.
- ❻ shows the last commit/revision (r381) of the Subversion repository being mapped and imported as a new commit/SHA-1 (750dd5...) into the local Git repository.
- ❼ shows that the current HEAD pointer position is pointing to the tip of the trunk remote branch (which is Subversion's r381).

Note that if you're using an older version of git, you may see a warning message resembling the following:

```
Using higher level of URL:
  http://google-...code.com/svn/google-notes =>
  http://google-...code.com/svn
W: Ignoring error from SVN, path probably does not exist:
  (160013): Filesystem has no item:
  Could not resolve path /google-notes for history
W: Do not be alarmed at the above message git-svn is just searching
  aggressively for old history.
  This may take a while on large repositories
```

As it states in the output, don't be alarmed. This is just git svn trying to access a higher-level directory than you've specified and being denied access. It's not a problem; this is why the message was removed from later versions.

You have successfully imported a Subversion repository into a local Git repository.

Discussion

Recall from technique 8 that you use git clone to initially download an entire repository. Subversion's usual equivalent to git clone is svn checkout, which only checks out the most recent revision. Because you're importing the Subversion repository into Git, you need to download all the previous revisions and all the branches and tags. Each new Git commit has metadata stored in the commit message containing the mapping between Git and Subversion commits.

You can see from listing 9.1 that git svn iterates through all the revisions in the Subversion repository and creates Git commits locally from the changes. This may take a long time on large repositories, but it only needs to be done once. After git svn clone has finished, you have a complete local copy of a Subversion repository.

Let's see an example of how git-svn formats a commit message.

Listing 9.2 Output: Subversion commit message in a Git repository

```
# git show -s --format=%B master
Log server and database in Lotus Notes errors (b/13059110)
Changes:
...
```

```
git svn-id:
  http://google-enterprise-connector-notes.googlecode.com/svn/trunk@381
  43735464-b96a-11de-8be4-e1425cda3908
```

SVN URL 1 points to the URL line.
UUID 2 points to the UUID line.

- ① is the full URL for this Subversion commit, including the Subversion branch (trunk) and revision (@381). The trunk is automatically mapped to the master Git branch.
- ② is a unique git svn identifier for this Subversion repository. This is used to ensure that the repository at the URL remains the same and is not replaced with another, which could cause errors when you try to update.

9.1 Subversion branches and tags

You may have noticed that the clone output sometimes mentions branches. Here's a sample that was cut from listing 9.1.

Listing 9.3 Output: clone branch detection

```
Found merge parent (svn:mergeinfo prop):
  ae01454731b1603701c59b78c3a2a2801eb4115f
r378 = 677696fd7befaa4212e760d62ab281780469ea00
  (refs/remotes/origin/3.2.x)
  M    projects/version.properties
r379 = 818430013a86360963676c8ff979cf59b64121ef
  (refs/remotes/origin/3.2.x)
Found possible branch point:
  http://google-enterprise-...googlecode.com/svn/branches/3.2.x =>
  http://google-enterprise-...googlecode.com/svn/tags/3.2.4, 379
Found branch parent: (refs/remotes/origin/tags/3.2.4)
  818430013a86360963676c8ff979cf59b64121ef
Following parent with do_switch
Successfully followed parent
```

1 Merge found points to the merge parent line.
2 Branch parent points to the SHA-1 of the found parent commit.
3 Commit branch points to the branch name.
4 Branch URL points to the branch URL.
5 Tag URL points to the tag URL.
6 Tag found points to the tag found line.
7 Tag parent points to the SHA-1 of the found tag commit.

- ① shows that git svn found one of the parent commits of a merge by looking at the svn:mergeinfo Subversion property on the commit.
- ② shows the SHA-1 of the found parent commit.
- ③ shows that the found parent commit is for the 3.2.x branch.
- ④ is the (abbreviated) URL for the branch that was used to create the tag commit.
- ⑤ is the (abbreviated) URL and revision number for the tag commit.
- ⑥ shows the parent commit that was found for the 3.2.4 tag commit.
- ⑦ shows the SHA-1 of the found tag commit.

Let's examine the structure of the Subversion repository by running `git branch --remote` to view all the Git remote branches created by `git svn`.

Listing 9.4 Output: remote branches

```
# cd google-notes/
# git branch --remote
origin/2.6.x
...
origin/3.2.x          ← ❶ 3.2 branch
origin/Notes-Connector
origin/dev           ← ❷ Work branch
origin/tags/1.0.0
origin/tags/2.8.4     ← ❸ Branch tag
origin/tags/2.8.4@273 ← ❹ Duplicate tag
...
origin/tags/3.2.4
origin/tags/builds
origin/trunk
```

- ❶ shows the stable 3.2 release branch named `3.2.x`. This will be used to create more patch tags in the 3.2 series: for example, `3.2.4`.
- ❷ is a named branch used for development work named `dev`.
- ❸ shows the branch for the `2.8.4` tag. Note that this has been imported as a branch and not a native Git tag. (This will be explained later.)
- ❹ shows the duplicate `2.8.4` tag named `2.8.4@273`. This is the case because it was revision 273, and the other `2.8.4` is at revision 274.

WHY ARE THERE TAGS IN THE BRANCH OUTPUT? You may have noticed that tags from `git svn` aren't the same as normal Git tags but instead are branches with a `tags/` prefix. This is because in Subversion, the only difference between a tag and a branch is that of principle. Generally you don't update tags in Subversion, but it's possible and has happened in this repository. There is a duplicated `2.8.4` tag (named `2.8.4@273`) because a commit was made to create the `2.8.4` tag, and another commit was made on it. This wouldn't be possible in Git; you'd need to use `git tag --force` to forcefully update the tag, and then the previous tag would be lost. This is the reason `git svn` doesn't import the Subversion tags as native Git tags. If you wanted to create native Git tags from these anyway, you could use `git branch --remote --list 'origin/tags/*'` to only show Subversion tags and then create Git tags manually. For example, to create the `3.2.4` tag, you could run `git tag 3.2.4 origin/tags/3.2.4`.

9.2 Subversion ignore rules

Recall from technique 21 that `.gitignore` files contain a list of patterns of paths for Git to ignore in a repository. Subversion uses the `svn:ignore` property on directories instead. These aren't imported by `git svn` into a `.gitignore` file automatically, because doing so would require adding a file to the repository.

You can export the `svn:ignore` property values to a `.gitignore` file by using the `git svn show-ignore` command.

Listing 9.5 Output: Subversion ignore rules

```
# git svn show-ignore

# /projects/
/projects/build
/projects/install
/projects/downloads

# /projects/notes-client/
...
```



1 Directory comment

2 Directory ignore

1 shows a `.gitignore` comment line (comments are prefixed with `#`) for the projects directory's `svn:ignore` property value.

2 shows an ignore rule for the projects directory to ignore a file or directory named `build`.

You can use the `git svn show-ignore` output to write a `.gitignore` file by running `git svn show-ignore > .gitignore`. The `>` redirects the command output from the terminal into the `.gitignore` file. You can then add and commit this file to the repository to share these rules with anyone else using `git svn`.

In some cases, you may not want people to know you're using `git svn` or not want to commit a `.gitignore` file to a Subversion repository. In this case, you could omit the `.gitignore` file or not add it to any commits, but this could become irritating when files aren't ignored. Instead, you can use `.git/info/exclude`, as you saw in technique 1, which operates like a local `.gitignore` file for a single repository. This file handily also uses the same syntax as `.gitignore`. You can write the ignore rules to it by running `git svn show-ignore > .git/info/exclude`.

9.3 Updating a Subversion repository

To update a Subversion repository, you need to use the `git svn` command; you can't use `git fetch` or `git pull` because `git svn` hasn't set up any remote Git repository references for you, because it doesn't use the same transport mechanism. When working locally, you use `git` commands as normal.

Figure 9.1 shows the `git svn` cycle you'll look at in this section. As in the local workflow in section 1.4, files are modified, added, committed, and can be checked out. But in comparison to technique 6, the remote repository is a Subversion repository and so requires different commands.

The equivalents to `git fetch` and `git pull --rebase` for Subversion repositories are `git svn fetch` and `git svn rebase`. There's no equivalent to `git pull` without `--rebase`. Git and Subversion handle merges differently, so it's important to avoid merge commits on updates because they won't (and shouldn't) be seen by other users of the Subversion repository.

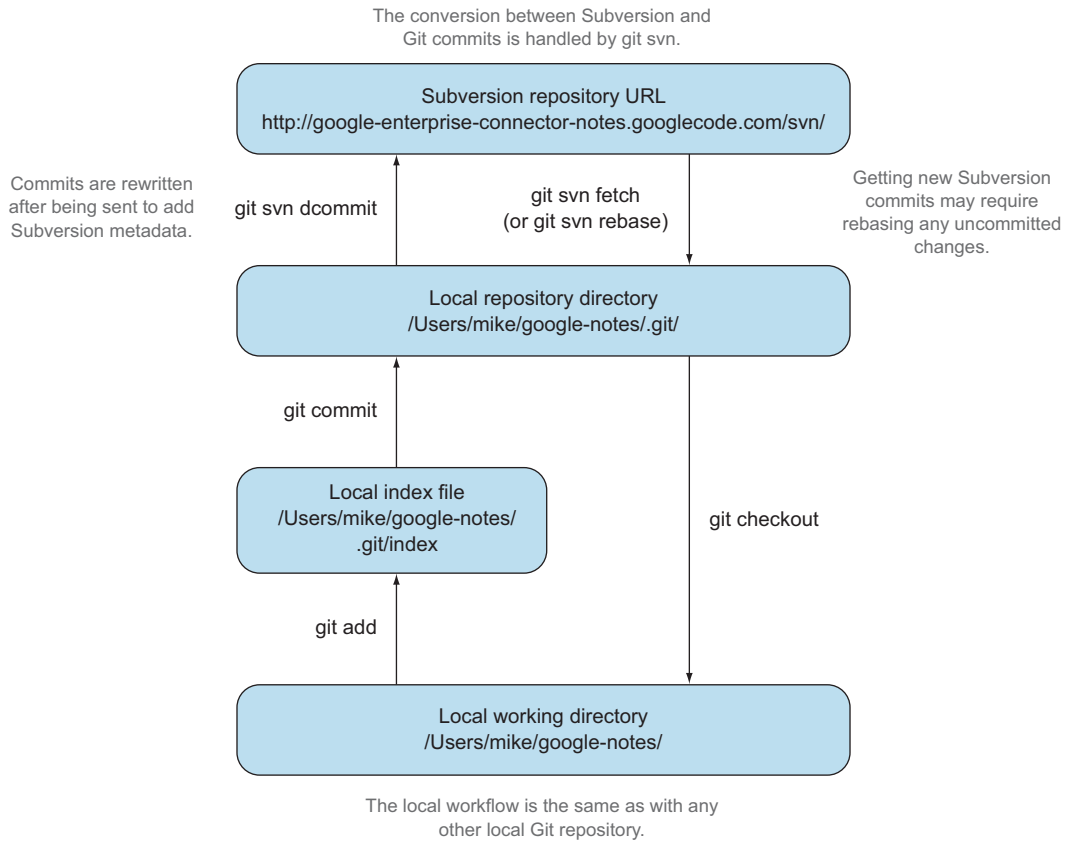


Figure 9.1 Git SVN add/commit/dcommit/rebase/checkout cycle

If you run `git svn rebase` on the master branch and there are no new commits, the output will be as follows.

Listing 9.6 Output: no new Subversion revisions

```
# git svn rebase

Current branch master is up to date.
```

If there was a single new revision (r2), the output might resemble this.

Listing 9.7 Output: one new Subversion revision

```
# git svn rebase

M    README.txt
r2 = 685b522aebec94dc75d725c34c092d9be5f3fc39 (remotes/origin/trunk)
First, rewinding head to replay your work on top of it...
Fast-forwarded master to remotes/origin/trunk.
```

Rebase begin (3) →

① Modified file

② New revision

④ Fast-forward

- ❶ shows that a file named README.txt was modified in the new revision.
- ❷ shows the new revision number (r2) and the new commit SHA-1 (685b52...).
- ❸ shows the beginning of the `git rebase` operation that `git svn rebase` is running to rebase any commits made on this branch on top of the newly received commits.
- ❹ shows that this `git rebase` was a fast-forward of the HEAD pointer to the latest new commit, because there were no local commits that needed to be rebased.

9.4 Subversion authors and committers

Let's look at the metadata of a commit imported from a Subversion repository.

Listing 9.8 Subversion commit metadata in a Git repository

```
# git show -s --format=short master
```

```
commit 750dd5077c4122bcc3a8a17a65088c0ebd85a2b6
Author: tdnguyen@google.com
<tdnguyen@google.com@43735464-b96a-11de-8be4-e1425cda3908>

Log server and database in Lotus Notes errors (b/13059110)
```

❶ Author name
❷ Author email

- ❶ shows an email address instead of the author name. This is the username of the user in the Subversion repository (which happens to be an email address in this case).
- ❷ shows the author email address. In `git-svn`, this is created from the username by appending to the username @ followed by the UUID for the Subversion repository.

It's possible to use a Subversion authors mapping file by passing the `--author-file` (or `-A`) flag to `git svn clone` when you first clone a Subversion repository. The authors file has the following syntax:

```
mikemcquaid = Mike McQuaid <mike@mikemcquaid.com>
```

If passed a valid file with this format, when `git svn` reads a new revision, it looks up the username in this file. If the username is `mikemcquaid`, it replaces the author (or committer) name and email address with those specified in the file. If it can't find an entry in the file, it stops the clone (or rebase), and you must add the new author's details to the file.

9.5 Viewing a Subversion repository in GitX

`git svn` creates a Git repository from a Subversion repository, so you can still use all the graphical tools you're used to. GitX also provides an additional Git SVN Revision column to display the Subversion revision number (see figure 9.2).

9.6 Migrating a Subversion repository to Git

With what you've already learned in this section (cloning a Subversion repository, creating real Git tags, and mapping authors), you can create a Git repository that con-

Short...	Subject	Author	Date	Git SVN Revision
750dd50	Log server an...	tdnguyen@google.com	7 March 2014 21:57	381
3679390	Tagging the 3...	tdnguyen@google.com	3 February 2014 1...	380
8184300	Set version to 3.2.4 on...	tdnguyen@google.com	3 February 2014 1...	379
677696f	Merge changes r355:377 on bra...	tdnguyen@google.com	3 February 2014 1...	378
ae01454	Update release notes for 3.2.4 re...	tdnguyen@google.com	28 January 2014 2...	377
08823b0	Bumped the SPI version to 3.2.4....	jlacey@google.com	28 January 2014 0...	376

Figure 9.2 GitX on import Subversion repository

tains all the information from a Subversion repository in the typical Git format. This may be useful if you want to migrate a project from Subversion to Git; you can import the entire history, migrate the tags, and `git push` it to a new repository. If you want to remove all references to the original Subversion repository, you can use `git filter-branch` (introduced in technique 47) to remove all the `git-svn` Subversion references from commit messages or otherwise reformat them.

Technique 57 Committing and pushing to an SVN repository from a Git repository

Remember that `svn commit` does the equivalent of a `git commit` and a `git push` to the remote server. The repository created by `git svn` is a normal Git repository, so you can change files and commit as you might do with any other Git repository. The only differences are when you want to push your changes to the Subversion repository, and if you want to interact with remote branches.

To push all the unpushed commits on the current branch to a Subversion repository, you use the `git svn dcommit` command.

Problem

You wish to commit and push changes to a Subversion repository.

Solution

- 1 Change directory to a Git SVN repository: for example,
`cd /Users/mike/GitSVN/.`
- 2 Make some changes to a file such as `README.txt` file, and commit them:
`git commit --message "README.txt: improve grammar." README.txt.`
- 3 Run `git svn dcommit`.

The output from these commands should resemble the following.

Listing 9.9 Output: Subversion push

```
# git commit --message "README.txt: improve grammar." README.txt

[master bcd0a70] README.txt: improve grammar.
1 file changed, 1 insertion(+), 1 deletion(-)

# git svn dcommit
```

← 1 New commit

```

Committing to http://svntest.com/svntest/ ...
M    README.txt
Committed r3
M    README.txt
r3 = da4cc700b6d5fe07ead532a34195b438680e7a71 (remotes/origin/trunk)
No changes between bcd0a70923a9b53cd98ccaeee1567ca95bb579c0 and
remotes/origin/trunk
Resetting to the latest remotes/origin/trunk

```

Commit diff 4 ← 2 Push success ← 3 New revision ← 5 Trunk reset

- 1 shows the commit subject and SHA-1 of the new commit.
- 2 shows that the new Subversion revision was committed successfully.
- 3 shows the new commit created from the new Subversion revision. Recall that commits contain their revision numbers and repository UUIDs, which requires rewriting the commit message. Also recall that rewriting the commit message changes the SHA-1 of a commit. As a result, this new commit SHA-1 doesn't match the SHA-1 in 1, although the actual changes are the same.
- 4 shows `git svn` checking that there are no differences between the commit that was just created and the commit the Subversion repository returned.
- 5 shows that the `HEAD` and `master` branch pointers are being updated to the new commit. The old commit is still accessible from before it was rewritten and the new commit was created with the Subversion metadata.

You have successfully committed and pushed changes to a Subversion repository.

Discussion

You can see that `git svn dcommit` also has to do some rewriting of commits, similar to `git svn rebase`. This is because the commit messages store additional metadata that can only be obtained from the Subversion server. The Subversion server may have had additional commits in the meantime, which means the revision number may differ from the last one seen. If this has happened, a rebase may need to be done by `git svn dcommit` after receiving the new commit from the server.

9.7 Local branching and tagging

Subversion doesn't have the concept of local branches or tags. If a branch or tag needs to be created in Subversion, the Subversion client has to speak to the server.

Because you have a local Git repository containing the contents of the Subversion repository, you're not bound by the same constraints. You can create local branches and tags and use them as you wish, and everything is fairly simple unless you want to send commits to or receive commits from the Subversion server.

Recall that both `git svn rebase` and `git svn dcommit` perform rebasing operations on updates. As a result, it becomes difficult to correctly handle merges between Subversion branches with `git svn`. You can read how to do this in `git svn --help` using the `--mergeinfo` flag, but I won't cover it in this book.

HOW SHOULD YOU COLLABORATE WHEN USING GIT SVN? I advise you to use local branches only for your own work, and not for collaboration with others. When you're finished with a local branch and wish to merge it, you should rebase the contents into the branch you wish to include it in. This means others won't see your merge commits, but you can still use the cheap local branches and history rewriting in Git. If you want to interact with Subversion remote branches or tags, you should instead use the `git svn branch` and `git svn tag` commands. These are copies of Subversion's `svn branch` and `svn tag` commands and take the same parameters and use the same syntax.

Technique 58 Accessing a GitHub repository with Subversion

So far, this chapter has been concerned with accessing Subversion repositories using Git. This assumes a development team that is mostly using Subversion, and a few users or single user using Git. Incidentally, this is how I learned Git originally; I worked on Subversion projects but used Git locally.

What if the situation were reversed, and the majority of people on the project wanted to use Git and a minority wanted to use Subversion? This is made better if you host your Git repository on GitHub, because GitHub provides a Subversion interface for every Git repository. Let's check out the `GitInPracticeRedux` repository from earlier chapters using `svn checkout`.

Problem

You wish to check out the `GitInPracticeRedux` repository from earlier chapters using Subversion.

Solution

- 1 Change to the directory where you want the new `GitInPracticeReduxSVN` repository to be created: for example, `cd /Users/mike/` to create the new local repository in `/Users/mike/GitInPracticeReduxSVN`.
- 2 Run `svn co https://github.com/GitInPractice/GitInPracticeRedux`. The output should resemble the following.

Listing 9.10 Partial output: checking out a GitHub repository with Subversion

```
# svn co https://github.com/GitInPractice/GitInPracticeRedux
A   GitInPracticeRedux/branches
A   GitInPracticeRedux/branches/inspiration
A   GitInPracticeRedux/branches/inspiration/.gitignore
A   GitInPracticeRedux/branches/inspiration/00-Preface.asciidoc
...
A   GitInPracticeRedux/branches/v0.1-release/00-Preface.asciidoc
...
A   GitInPracticeRedux/tags/v0.1/00-Preface.asciidoc
...
A   GitInPracticeRedux/trunk/00-Preface.asciidoc
A   GitInPracticeRedux/trunk/01-IntroducingGitInPractice.asciidoc
```

Diagram illustrating the output of the `svn co` command, showing the repository structure and the checked-out files:

- 1 Inspiration branch**: Points to the `inspiration` branch.
- 2 v0.1-release branch**: Points to the `v0.1-release` branch.
- 3 v0.1 tag**: Points to the `v0.1` tag.
- 4 Trunk**: Points to the `trunk` directory.


```
A      GitInPracticeRedux/trunk/02-AdvancedGitInPractice.asciidoc
Checked out revision 26.
```

← 5 Latest revision

- ❶ shows the 00-Preface.asciidoc file in the inspiration branch.
- ❷ shows the 00-Preface.asciidoc file in the v0.1-release branch.
- ❸ shows the 00-Preface.asciidoc file in the v0.1 tag.
- ❹ shows the 00-Preface.asciidoc file in trunk (which is actually the renamed master branch).
- ❺ shows the latest revision number for the repository (r26).

You have checked out the GitInPracticeRedux repository using Subversion.

Discussion

As you can see, the Git repository has been transformed into the traditional Subversion layout with trunk, branches, and tags folders in the root. Typically you'd use `svn co https://github.com/GitInPractice/GitInPracticeRedux/trunk` instead and switch to the current branch of choice using `svn switch`.

You can use `svn commit`, `svn branch`, and any other Subversion commands with this repository. They're mapped on the GitHub servers into the corresponding Git commands.

You can read more about GitHub Subversion integration at <https://help.github.com/articles/support-for-subversion-clients>. The current implementation-specific details are beyond the scope of this book and not necessary for typical use.

If you're already using or considering GitHub, I'd strongly recommend using the GitHub repository through Subversion rather than a Subversion repository through `git-svn`. This is because Subversion's functionality is effectively a subset of Git's functionality, so using GitHub's Subversion support won't limit Git users as much (if at all) compared to Git users using `git svn`. For example, you can happily merge branches using Git and push them when using GitHub's Subversion integration, whereas when using `git-svn`, as mentioned in technique 57, you should do branch merges using Subversion's tools instead.

If you're not using GitHub, there are tools such as SubGit (<http://subgit.com>) that are beyond the scope of this book but may enable you to work in teams with some users using Git and others using Subversion.

9.8 Summary

In this chapter, you learned the following:

- How to use `git svn clone` to import an existing Subversion repository
- How to use `git svn rebase` to fetch from, and `git svn dcommit` to push to, an existing Subversion repository
- How to use `svn checkout` to check out GitHub repositories using Subversion

10

GitHub pull requests

This chapter covers

- Making pull requests from branches on the same repository
- Making pull requests from branches on a forked repository
- Interacting with forks and pull requests from the command line
- Merging a pull request from a forked repository

GitHub provides more than just Git hosting that you can `git push` to and `git fetch` from; it also provides features that allow teams of individuals to collaborate on software projects in ways beyond those provided by Git itself. Two of these features are *pull requests* and *forks*. In this chapter you'll learn about GitHub's pull requests.

10.1 What are pull requests and forks?

One of the core components of most software project collaboration is some sort of issue (or bug) tracker, such as JIRA, FogBugz, Pivotal Tracker, or Trac. GitHub provides an issue tracker for every repository known as *Issues*. In GitHub's Issues, there are subject and body fields, an open or closed state, labels, and comments.

There is a special type of issue known as a *pull request* (*PR*). This is an issue on a project with an attached branch of code changes. Pull requests are used to request

that someone with commit access to the repository merge the code changes from the pull-request branch into their the main repository. Once a PR has been received, you can view the changes that have been requested, and anyone who can view the PR can make comments either inline on the commit or generally on the PR. If there are comments that require changes to be made, they can be committed and pushed to the pull-request branch, and the pull request is updated with these changes. Once the PR's changes have reached an acceptable standard to be included in the repository, you can use the Merge button on the GitHub website to merge the requested branch into the repository.

Let's say you're creating a new web application and want the `master` branch to be merged regularly to the web server. You want to ensure that any branches merged into the `master` branch are always checked by at least one other developer before they're merged (to be sure nobody accidentally deploys bad code to the server). To do this, you can request that anyone using the project create a pull request rather than merging directly to `master` and that they receive an OK from at least one other developer before merging to `master`. If the changes aren't yet OK, then someone can add more commits to the pull request, and it will be merged when it has been OK'd. Adopting this approach will result in higher-quality code being merged to `master` and fewer silly mistakes, because any new changes will be checked by at least one person other than the original author.

Recall from technique 8 that you use `git clone` to initially download an entire repository. A *fork* is similar to a clone, but rather than download the repository to your local machine, it creates a clone of a repository on GitHub's servers. This allows you to make any changes you wish to your fork even if you don't have commit access to the original repository. Anyone with read access to a repository can fork it.

PRs and forks can be combined, so you can find a repository you wish to make changes to, fork it, and make changes in your local fork. You can then create a PR to request the changes on the branch in your fork be merged into the original repository.

Let's return to the web application example. Let's say you have an open source library that your web application depends on, and you've found a bug in it. You want to fix this bug and share it with the open source library maintainers so that they can provide the fix to everyone. But you've never worked on this project before, and the maintainers don't know or trust you yet—thus they're unlikely to give you commit access to the project so you can fix the bug. Instead, you can fork the repository, make the changes, commit and test them, and then create a pull request from your forked repository to the original repository. This allows the maintainers of the project to review your proposed changes, make any comments, and merge them when they're ready. This provides a code-review workflow similar to that when using pull requests without forks, but it means anyone can suggest changes without having the ability to merge them.

You can read more about using pull requests and forking repositories in GitHub's official help pages at <https://help.github.com/articles/using-pull-requests> and

<https://help.github.com/articles/fork-a-repo>. I don't go into more detail here, because these pages are kept up to date with any changes made to GitHub.

10.2 **Interacting with GitHub from the command-line: gh**

You've been working mostly from the command line in this book, and PRs are sometimes a core part of Git workflows, so it would be useful if you could also create PRs from the command line. Fortunately, a tool called `gh` lets you interact with GitHub from the command line (through the GitHub API). It's available for download and provides installation instructions at <https://github.com/jingweno/gh>.

The `gh` command-line tool allows various interactions with GitHub by providing a wrapper on top of Git's interface and adding new commands and improving the interaction of some Git commands with GitHub. For example, instead of running `git clone https://github.com/GitInPractice/GitInPracticeRedux`, you can run `gh clone GitInPractice/GitInPracticeRedux` to perform the same action of cloning a GitHub repository to your local machine.

The `gh` tool can also be used to create PRs from the command line. Although this accomplishes the same end result as the web interface and is slightly less flexible, I generally prefer it, because it fits better into the command-line-driven workflow of using the Git to interact with a project's repository.

Technique 59 Making a pull request in the same repository: gh pull-request

In my experience, most teams that work on private repositories tend to use a single repository rather than many different forks. Pull requests can still be used on a private repository; even although everyone with read access also has commit access, PRs are still useful for performing code review before changes are merged into a repository.

To create a PR, you need to create a local branch of the changes you want to be merged into another branch (`master`, by default), push the local branch to a remote branch, and use `gh` to create a pull request.

Problem

You wish to create a GitHub pull request from the command line.

Solution

- 1 Change to the directory containing your repository: for example,
`cd /Users/mike/GitInPracticeRedux/`.
- 2 Create a new branch from the master branch by running a command like `git checkout -b pr-test master`.
- 3 Make some changes to a file—for example, `00-Preface.asciidoc`—and commit them by running `git commit 00-Preface.asciidoc`. Ensure that there are multiple lines in the commit message: a commit subject and commit body.

- 4 Push the local branch to a remote branch by running `git push --set-upstream origin pr-test`.
- 5 Create a pull request by running `gh pull-request`, and accept the default pull request message (which should match the commit message).

The output for these commands should resemble the following.

Listing 10.1 Output: creating a pull request with `gh`

```
# git checkout -b pr-test master
Switched to a new branch 'pr-test'
# git commit 00-Preface.asciidoc
[pr-test 071d468] Preface: use Praxis instead of Paris.
 1 file changed, 1 insertion(+), 2 deletions(-)
# git push --set-upstream origin pr-test
Counting objects: 10, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 348 bytes | 0 bytes/s, done.
Total 3 (delta 2), reused 0 (delta 0)
To https://github.com/GitInPractice/GitInPracticeRedux
 * [new branch]      pr-test -> pr-test
Branch pr-test set up to track remote branch pr-test from origin.
# gh pull-request
https://github.com/GitInPractice/GitInPracticeRedux/pull/1
```

- ① shows the creation of the new local branch. This is used to commit the changes for the new pull request.
- ② shows the new commit on the `pr-test` branch. It was created with a commit message subject and body.
- ③ shows the new remote branch that was created to be used for the new pull request.
- ④ shows the URL for the new pull request created by the `gh` tool using the GitHub API. It requests the merge of the `pr-test` branch into the remote repository's master branch.

You have created a new pull request from the command line using the `gh` tool.

Discussion

HOW DO YOU USE BRANCHES WITH PULL REQUESTS? Each pull request you create should use a new, non-master branch. Because each pull request tracks the status (and any new commits) for a particular remote branch, you need to ensure that each branch is used for a separate pull request to avoid situations like adding a new commit to one pull request and having it show up in another. You should also avoid creating pull requests from the master branch, because this is generally the branch you will want to merge to. Additionally,

GitHub sometimes doesn't update the master branch if you push new commits to it after creating the pull request, so you'd need to create a new pull request for every change that needs to be made. This is less than ideal, because you lose all the existing context and comments.

Now that a pull request has been created, you can view it in the GitHub web interface. Figure 10.1 shows the new pull request created on GitHub. *gh* defaulted the pull-request message to that of the single commit in this pull request. You can see that the commit message subject was used for the title of the pull request and the commit message body was used as the initial comment. Additionally, the master branch was used as the *base branch*, which is the branch into which the PR requests the changes be merged. The changes that should be merged in are those from the *pr-test* branch, which is known as the *head branch*.

Essentially, the pull request has created a remote branch named *pr-test*. You've requested that someone merge it into the *master* branch and discuss any changes that need to be made.

The *gh* tool also accepts the *-b* and *-h* parameters, which can be followed with a branch name to change the base and head branches, respectively. These can be passed either a branch name such as *pr-test*; a branch name and GitHub user/organization name such as *GitInPractice:pr-test*; or a GitHub user/organization name, repository name, and branch name, as in *GitInPractice/GitInPractice-Redux:pr-test*.

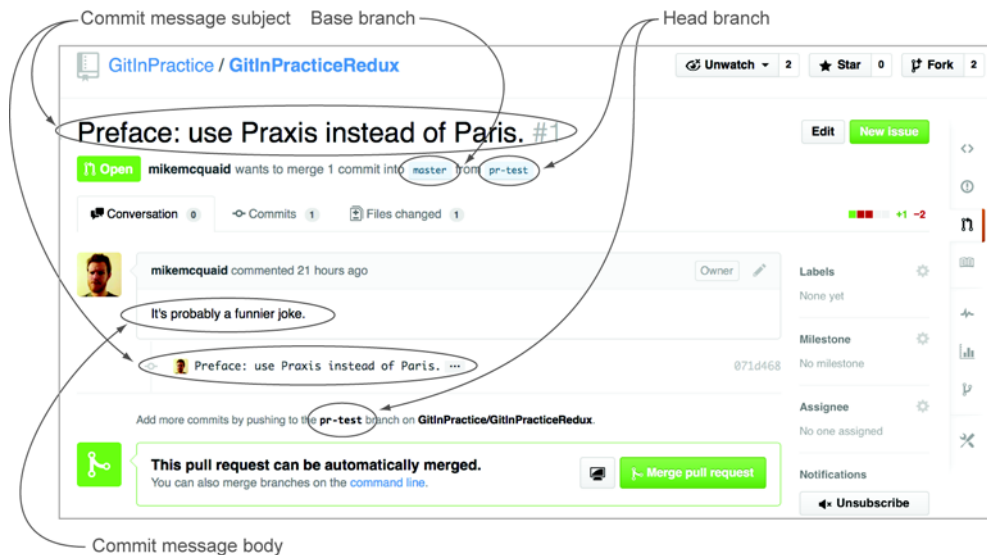


Figure 10.1 New pull request

HOW CAN YOU VIEW A PULL REQUEST WITHOUT WHITESPACE CHANGES? The Files Changed pane on a pull request shows a diff of the changes in that pull request. Some lines may have changes to whitespace that you don't care about. In this case, you can avoid them by appending `?w=1` to the Files Changed URL: for example, <https://github.com/GitInPractice/GitInPracticeRedux/pull/1/files?w=1>.

Technique 60 *Making a pull request from a forked repository: gh fork*

If you want to commit to an open source software project that you don't have commit access to, you'll want to create a pull request so others can review your changes before they're merged (because open source doesn't mean letting anyone commit to any repository at any time). To do this, you create your own repository to which you can make commits and push branches, and from which you can request pull requests. As you may recall from section 10.1, you can fork any repository you have read access to (which includes all public, open source repositories).

Problem

You wish to fork a repository and create a pull request from that fork from the command line.

Solution

- 1 Change to the directory containing your repository: for example, `cd /Users/mike/GitInPracticeRedux/`.
- 2 Create a new fork by running `gh fork`.
- 3 Create a new branch from the master branch by running a command like `git checkout -b credits master`.
- 4 Make some changes to a file—for example, `01-IntroducingGitInPractice.asciidoc`—and commit them by running `git commit 01-IntroducingGitInPractice.asciidoc`. Ensure that there are multiple lines in the commit message: a commit subject and commit body.
- 5 Push the local branch to a remote branch by running, for example, `git push --set-upstream origin pr-test`.
- 6 Create a pull request by running `gh pull-request`, and accept the default pull-request message (which should match the commit message).

The output for these commands should resemble the following.

Listing 10.2 *Output: creating a fork and a pull request with gh*

```
# gh fork
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
```

```

From https://github.com/mikemcquaid/GitInPracticeRedux
* [new branch]      inspiration -> mikemcquaid/inspiration
* [new branch]      master      -> mikemcquaid/master
* [new branch]      pr-test     -> mikemcquaid/pr-test
* [new branch]      v0.1-release -> mikemcquaid/v0.1-release
new remote: mikemcquaid
# git checkout -b credits

Switched to a new branch 'credits'

# git commit 01-IntroducingGitInPractice.asciidoc

[credits e9d27c7] Chapter 1: attribute quote.
1 file changed, 2 insertions(+), 2 deletions(-)

# git push --set-upstream origin credits

Counting objects: 10, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 348 bytes | 0 bytes/s, done.
Total 3 (delta 2), reused 0 (delta 0)
To https://github.com/mikemcquaid/GitInPracticeRedux
* [new branch]      credits -> credits
Branch credits set up to track remote branch credits from origin.

# gh pull-request

https://github.com/GitInPractice/GitInPracticeRedux/pull/2

```

- ① shows that the repository was forked on GitHub, and a new remote repository was added with the username of the fork (mikemcquaid in this case) and then fetched.
- ② shows the creation of the new local branch.
- ③ shows the new commit on the `credits` branch.
- ④ shows the new remote branch that was created.
- ⑤ shows the URL for the new pull request. It's requesting the merge of the `credits` branch from the <https://github.com/mikemcquaid/GitInPracticeRedux> forked repository into the `master` branch of the <https://github.com/GitInPractice/GitInPracticeRedux> main repository.

You have created a fork and a pull request from it from the command line.

Technique 61 Merging a pull request from the same repository

Merging a pull request from a non-forked repository is easy. You can either click the Merge Pull Request button (as shown in figure 10.1) or merge the branch as you would any other. Note that the Merge Pull Request button always performs a non-fast-forward merge (it always produces a merge commit), so let's do that here too.

Problem

You wish to merge a pull request from the command line.

Solution

- 1 Change to the directory containing your repository: for example,
`cd /Users/mike/GitInPracticeRedux/`.
- 2 Check out the master branch by running `git checkout master`.
- 3 Ensure that all the remote branches are up to date by running `git fetch`.
- 4 Merge the remote `pr-test` branch into the master branch by running `git merge --no-ff origin/pr-test`.
- 5 Push the updated master branch with `git push`.
- 6 Delete the now-merged `pr-test` branch by running `git push --delete origin pr-test`.

The output for these commands should resemble the following.

Listing 10.3 Output: merging a pull request

```
# git checkout master
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.
# git fetch
# git merge --no-ff origin/pr-test
Merge made by the 'recursive' strategy.
 00-Preface.asciidoc | 3 +--
 1 file changed, 1 insertion(+), 2 deletions(-)
# git push
Counting objects: 1, done.
Writing objects: 100% (1/1), 241 bytes | 0 bytes/s, done.
Total 1 (delta 0), reused 0 (delta 0)
To https://github.com/GitInPractice/GitInPracticeRedux.git
 cc206b5..7a19d89  master -> master
# git push --delete origin pr-test
To https://github.com/GitInPractice/GitInPracticeRedux.git
 - [deleted]          pr-test
```

← ❶ Branch checkout

← ❷ Merge commit

← ❸ Branch push

← ❹ Branch delete

- ❶ shows the checkout of the master branch. This is required, because to merge into the master branch, you first need to have the master branch checked out.
- ❷ shows the new merge commit created by the non-fast-forward commit. Remember, this was run with `--no-ff` to guarantee creating a merge commit (which matches the behavior of the Merge Pull Request button).
- ❸ shows the new merged commits being pushed to the remote master branch.
- ❹ shows the deletion of the now merged (and therefore unneeded) `pr-test` branch.

You have successfully merged a pull request from the command line.

Discussion

Let's look at the pull request on GitHub. Figure 10.2 shows the state of the merged pull request after these changes. You'll notice that after the push, the pull request is automatically closed, because it has detected that you've merged the contents of the branch remotely.

HOW CAN YOU CLOSE A PULL REQUEST WITHOUT A MERGE OR ISSUE FROM A COMMIT? If you prefer to cherry-pick or rewrite some of the commits, the SHA-1 may change. This may not be detected automatically by GitHub as a merge, and therefore the issue may not be closed automatically. If you wish to ensure that any modified commit automatically closes the pull request, you can use `git rebase --interactive` or `git commit --amend` to change the commit message for one of the commits to include text such as `Closes #1`. The first pull request that was created in the example was numbered #1. This magic string in a commit message is detected by GitHub and indicates that when this commit is merged to the `master` branch on the main repository, it should close the pull request or issue numbered #1. You can read more about this in GitHub's help at <https://help.github.com/articles/closing-issues-via-commit-messages>.

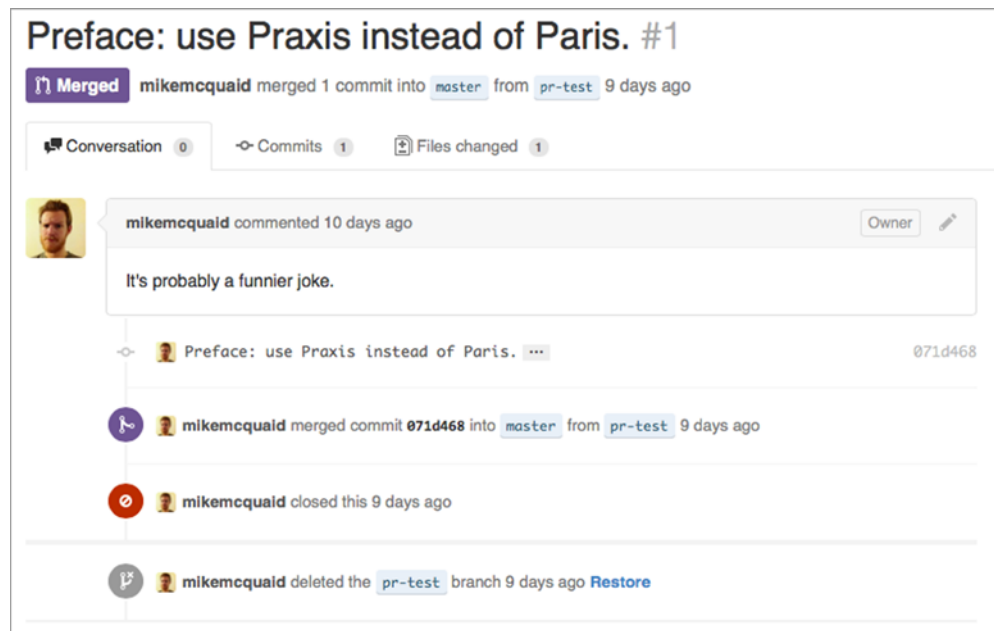


Figure 10.2 Merged pull request

Technique 62 *Merging a pull request from a forked repository: gh merge*

Merging a pull request from a forked repository is more involved. You can click the Merge Pull Request button, but what if you want to merge the branch from the command line? You could manually add the forked repository as a remote repository and merge it like before. Instead, though, let's use the handy `gh` tool again to make things a bit easier.

To simulate the typical open source maintainer approach where new forks aren't already added as remote repositories, let's start by removing the fork's remote from the local Git repository by running `git remote rm mikemcquaid` (which won't produce any output). This leaves the remote intact on GitHub, but it's no longer on the local repository.

Problem

You wish to merge a pull request from a forked repository from the command line.

Solution

- 1 Change to the directory containing your repository: for example, `cd /Users/mike/GitInPracticeRedux/`.
- 2 Check out the master branch by running `git checkout master`.
- 3 Merge the remote pull request into the master branch by running `gh merge https://github.com/GitInPractice/GitInPracticeRedux/pull/2`.
- 4 Push the updated master branch with `git push`.

The output for these commands should resemble the following.

Listing 10.4 Output: merging a pull request from a forked repository

```
# git checkout master

Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.

# gh merge https://github.com/GitInPractice/GitInPracticeRedux/pull/2

From https://github.com/mikemcquaid/GitInPracticeRedux
* [new branch]      credits    -> mikemcquaid/credits
Merge made by the 'recursive' strategy.
 01-IntroducingGitInPractice.asciidoc | 4 +---
 1 file changed, 2 insertions(+), 2 deletions(-)

# git push

Counting objects: 12, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (5/5), 620 bytes | 0 bytes/s, done.
Total 5 (delta 3), reused 0 (delta 0)
To https://github.com/GitInPractice/GitInPracticeRedux.git
```

← 1 Branch checkout

← 2 PR merge

```
7a19d89..77f848d master -> master
```

← 3 **Branch push**

- ❶ shows the checkout of the master branch.
- ❷ shows the new merge commit created by the pull-request merge.
- ❸ shows the new PR commits being pushed to the remote master branch.

You have successfully merged a pull request from a forked repository from the command line.

10.3 Summary

In this chapter you learned the following:

- How pull requests are used to request the merge and review of branches
- How forks are used to request pull requests on repositories without commit access
- How to create a new pull request using `gh pull-request`
- How to merge a pull request using `git merge` or `gh merge`

11

Hosting a repository

This chapter covers

- Creating a repository in a format for hosting a server
- Mirroring an existing repository for hosting a server
- Sharing a repository with a local network
- Viewing a repository in a web browser
- Providing advanced Git hosting with other software

You saw in technique 6 how to push to remote repositories, provided by various organizations on the internet, as a way of sharing your Git repositories with others. Sometimes, though, you may wish to host a remote Git repository on a machine you control—for example, to

- Temporarily share a Git repository on your machine with another computer on the same network
- Keep a full backup of an external Git repository on a server you control
- Provide an internal mirror of an external Git repository to allow faster transfer speeds
- Host a Git repository on a server entirely under your own control

I only use the commands in this chapter to view a local repository or share a local repository with computers on the same network (rather than over the internet). Although you'll see more advanced tools in section 11.1, this chapter focuses on

understanding the features provided by Git itself and how they work, rather than more advanced Git hosting solutions. In essence, this chapter is a guide for software engineers to understand how Git repositories can be shared and not a guide for system administrators on how to set up Git servers. Unless you're an experienced system administrator, I'd advise using an external Git hosting provider (such as GitHub) rather than running your own Git servers.

Technique 63 *Initializing a local repository in a server hosting format: git init --bare*

The Git repositories you've seen throughout this book have all had a similar structure: the working directory contains a checkout of the files in the current branch and a .git subdirectory that contains the repository data. For example, if you had a `GitInPracticeNonBare.git` repo, its contents might resemble figure 11.1.

You can see that it has only the `README.md` file and the .git repository data subdirectory in the root. Git stores data in a highly space-efficient format. The checked-out files in a repository's working directory may sometimes take up more space than the compressed version of all files stored in the .git directory!

On a server, the working directory should never be used directly, so it's better to not create one at all. Because you'll be sending/receiving Git objects to/from various Git clients with `git push`, `git fetch`, and `git pull`, you don't need to have the actual files checked out on disk. A Git repository without a working directory is known as a *bare repository*. The major difference compared to the first repository you created in technique 1 is that this repository isn't in a .git directory.

Let's create a bare repository and look at its contents.

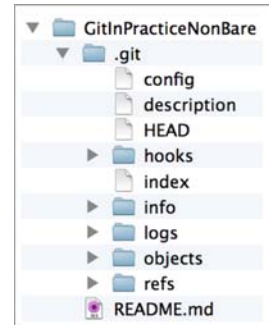


Figure 11.1 Typical repository layout

Problem

You wish to create a bare Git repository.

Solution

- 1 Change to the directory you wish to contain your new repository directory: for example, `cd /Users/mike/`.
- 2 Run `git init --bare GitInPracticeBare.git`. The output should resemble the following.

Listing 11.1 Output: initializing a bare repository

```
# git init --bare GitInPracticeBare.git
```

```
Initialized empty Git repository in /Users/mike/GitInPracticeBare.git/ ← 1
```

1 shows the new directory that was created for the bare Git repository.

You have successfully created a new bare Git repository.

Discussion

Bare repositories don't allow new commits to be created locally; they must be pushed from another repository.

HOW SHOULD YOU NAME BARE REPOSITORIES? When creating bare repositories, it's good practice to name them with the extension *.git* to make it clear that they're bare.

Let's look at the layout of a bare repository; see figure 11.2. You can see that the repository data that was stored in *.git* in figure 11.1 is instead in the root in figure 11.2. This means the root of the repository is effectively the *.git* directory (which is why it's named *GitInPracticeBare.git*). The bare repository is missing an index file and a logs directory and instead has a *packed-refs* file. These differences are internal Git files that are used for repositories; don't worry about their contents (although you'll see the *packed-refs* file in technique 64).

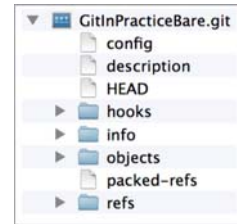


Figure 11.2 Bare repository layout

To clone this Git repository into a non-bare repository on the same machine, run `git clone` with the path on disk to the repository and the new, non-bare repository name as arguments: for example, `git clone /Users/mike/GitInPracticeBare.git GitInPracticeNonBare`. In this case, the output will resemble the following:

```
Cloning into 'GitInPracticeNonBare.git'...
warning: You appear to have cloned an empty repository.
done
```

The new repository has been created, but it's empty—it contains no commits. If you add a file, commit, and push it, it will be pushed to the bare repository and both repositories will be non-empty. Note that you'd never usually clone a bare repository on the same machine but would instead create a bare repository on one machine and clone it on another.

Personally, I've only used bare repositories to create mirrors of existing repositories on a server I control. I didn't care about having the files checked-out into the working directory; I just wanted a copy of all the repository data. Let's learn how to use Git to create an exact mirror of another repository.

Technique 64 *Mirroring a repository: `git clone --mirror`*

There are times when you wish to host a new Git repository that's a mirror of another—a functionally identical copy. This could be for backup, providing a local cache for increased speed, or moving a repository to another location. Recall from technique 8 that `git clone` creates a clone of the repository locally with all commits, branches, and tags that are in the repository you've cloned from.

If you `git clone` a repository with a branch named `testing`, your new, local clone will contain a remote branch named `origin/testing`. But what if you wanted this to create not only the `origin/testing` remote branch but also a local branch named `testing`? In this case, you'd use the `--mirror` flag for `git clone`, which creates local branches for all remote branches it finds (or, in Git terminology, *matches* all refs locally). This is useful when you want to create an exact copy of another repository (a mirror) so others can clone from it and get the same results as cloning from the original repository, or to keep as a backup. Recall the discussion of *forks* on GitHub from section 10.1. `git clone --mirror` is effectively what GitHub does when you fork a repository: it makes a complete copy that can be modified without changing the original repository and creates all the same branches.

Let's use `git clone --mirror` to set up a local mirror of the `GitInPracticeRedux` repository.

Problem

You wish to mirror an existing remote repository.

Solution

- 1 Change to the directory you wish to contain your new repository directory: for example, `cd /Users/mike/`.
- 2 Run `git clone --mirror https://github.com/GitInPractice/GitInPracticeRedux.git`. The output should resemble the following.

Listing 11.2 Output: cloning a mirror

```
# git clone --mirror
https://github.com/GitInPractice/GitInPracticeRedux.git
```

```
Cloning into bare repository 'GitInPracticeRedux.git'... ← ❶ Bare repository
remote: Reusing existing pack: 79, done.
remote: Counting objects: 1, done.
remote: Total 80 (delta 0), reused 1 (delta 0)
Unpacking objects: 100% (80/80), done.
Checking connectivity... done.
```

- ❶ shows that `git clone --mirror` creates a bare repository when it creates a mirror. This is because `--mirror` is used only when hosting a repository for other repositories to pull from.

You have mirrored the existing `GitInPracticeRedux` repository.

Discussion

As explained in technique 8, `git clone` can take the following tags:

- *No flags*—Creates a normal (non-bare) repository with remote branches
- `--bare` flag—Creates a bare repository with remote branches

- `--mirror` flag—Creates a bare repository with remote branches and local branches for every remote branch

Let's examine the contents of the `GitInPracticeRedux.git/packed-refs` file:

```
# pack-refs with: peeled fully-peeled
ca74d2b7c4dd15a260e68c6ff3552c64041aacdc refs/heads/inspiration
a9e150fb17301eed6c31aa984411effdab8f3fec refs/heads/master
a8200e1407d49e37baad47da04c0981f43d7c7ff refs/heads/v0.1-release
071d468df295c3866054763250a1344e44f8c3be refs/pull/1/head
75f9dd1ddc24e1fd9e58b8443f7f0176cf7bd2e7 refs/pull/1/merge
e9d27c7df49c07cb2325356ab9a76f90d9f179ae refs/pull/2/head
e6e9208372f3784686499430fec547c20dad6139 refs/pull/2/merge
725c33ace6cd7b281c2d3b342ca05562d3dc7335 refs/tags/v0.1
```

Branch
Pull request
← Tag

The `packed-refs` file contains all the *packed* refs (refs in Git's format for data internal and external transfer) that were fetched from the `GitInPracticeRedux` repository. It contains all the created branches, pull requests, and tags that were created in this repository. These will now be shared with any other repositories that clone this one.

Technique 65 *Sharing a repository with other users on the same network: git daemon*

Now that you've seen how to create bare repositories suitable for a server to share with other Git repositories, let's learn how to serve these to other Git clients. In technique 63, you saw a repository being cloned from another path on the disk. Although this would be one way of sharing a repository over the network with Git (giving someone access to your disk with, say, a network share), it's not very efficient, because it uses multiple protocols: the SMB protocol to share the files over the network, and Git's interaction with the packed repository. Instead, a Git server allows Git to interact natively in its own format and `git://` protocol, which transfers repository data in a format similar to how it's stored locally and defaults to using port 9418.

Git provides a simple command for basic repository hosting named `git daemon`. It provides no user authentication or encryption and only supports the `git://` protocol (rather than the `https://` you've used throughout this book, or `ssh://`, which uses SSH access). These protocols are fairly interchangeable; which one you pick will depend mostly on whether you need to use HTTP proxies or web servers (for the `https://` protocol), user authentication using SSH (for the `ssh://` protocol), or no authentication (for the `git://` protocol). This command may be too limited for some cases, but it's great for this technique.

Problem

You wish to share a repository with other users on the same network.

Solution

- 1 Change directory to the Git repository: for example,
`cd /Users/mike/GitInPracticeRedux.git/.`

- 2 Run `git daemon --verbose --base-path=. --export-all`. The output should resemble the following.

Listing 11.3 Output: daemon

```
# git daemon --verbose --base-path=. --export-all
```

← 1 **Daemon arguments**

```
[72938] Ready to rumble
```

← 2 **Process ready**

- 1 shows the daemon command and the list of arguments required to export the Git repository. These are elaborated on in the discussion section.
- 2 shows the process ID (72938), that the process has started successfully, and that it's ready to receive clients.

Now that you have `git daemon` running, open another terminal window and clone this repository from a client with `git clone git://localhost/`:

```
# git clone git://localhost/ GitInPracticeReduxDaemon
```

← 1 **Local server**

```
Cloning into 'GitInPracticeReduxDaemon'...
remote: Counting objects: 78, done.
remote: Compressing objects: 100% (71/71), done.
remote: Total 78 (delta 26), reused 0 (delta 0)
Receiving objects: 100% (78/78), 7.80 KiB | 0 bytes/s, done.
Resolving deltas: 100% (26/26), done.
Checking connectivity... done.
```

- 1 shows that you're using localhost to access the Git daemon on the same machine you're hosting it on. It has cloned the repository as expected into a new directory on the same machine. If you wanted to clone this from another machine, you'd replace localhost in the command with the IP address of the machine hosting the daemon on the network: for example, `git clone git://192.168.0.123/`.

If you view the daemon output again, you'll see that some lines have been added:

```
[72984] Connection from [::1]:52891
```

← **Client connection**

```
[72984] Extended attributes (16 bytes) exist <host=localhost>
```

← **Attribute exposure**

```
[72984] Request upload-pack for '/'
```

← **Repository upload**

```
[72938] [72984] Disconnected
```

← **Client disconnect**

These lines show that the Git client connected to the server, the repository exposed some attributes to the client, the client requested that the server upload its contents to the client, and the client then disconnected from the server.

You have successfully shared a repository over the network.

Discussion

`git daemon` can take some parameters to customize its behavior:

- `--verbose`—Outputs more verbose log details to the terminal about incoming Git client connections and access successes and failures. It's useful when hosting a server to enable this for debugging.
- `--base-path=.`—Indicates what path should be used as the server root. In this case, you only hosted a single repository, so you set the root to the base directory of the repository. If you wanted to host a directory that contained multiple repositories (such as `fish.git` and `cat.git`), you could specify the directories, and then they could be accessed by name (`git clone git://localhost/fish.git` or `git clone git://localhost/cat.git`). I tend to only use `git daemon` to share a single repository, so I use `--base-path=.`
- `--export-all`—Tells Git to allow access to all Git repositories under the base path. Without this argument, by default `git daemon` only allows access to repositories that have a `git-daemon-export-ok` file in the repository root (the root for bare repositories and `.git` for non-bare repositories). I tend to use this, because I use `git daemon` so infrequently and only on repositories I explicitly, currently want to share.
- `--enable=receive-pack`—Allows write access to the repository. By default, `git daemon` only allows read access (provided by `upload-pack`) to repositories unless this flag is provided. It's not recommended that you provide write access to non-bare repositories, because it would be undesirable for remote users to be able to change the contents of your local branches.
- *directory*—Needed if you wish to host a non-bare repository. In this case, you'd `cd` into the directory as normal but add a `./.git` argument specifying that you want to share the `.git` directory. For example, you might run `cd /Users/mike/GitInPracticeRedux && git daemon --verbose --base-path=. --export-all ./.git`. I use this when temporarily hosting non-bare repositories that I'm working with on my local machine with others.

Technique 66 *Displaying a repository in a browser: git instaweb*

Now that you've shared your repository on disk with other users, it would be useful if you could provide a basic web interface to go along with your `git daemon`. Git provides a basic web interface named `gitweb` that can be hosted by a local web server.

HOW CAN YOU INSTALL GITWEB? Gitweb is usually installed as part of the default Git installation (and is in all the official Git installers). If it hasn't been, you'll need to install it separately. This can be done by installing `gitweb` (or similar) with your package manager: for example, on Debian/Ubuntu, run `apt-get install gitweb`.

Git provides the `git instaweb` command to host your local repository using the `gitweb` interface. To run this, you must have a web server installed on your machine. If you're using OS X, you can use `WEBrick`, which is a simple web server provided with Ruby (which is provided with OS X). If you're on Linux, you can install Ruby with your

package manager: for example, on Debian/Ubuntu, run `apt-get install ruby` (you'll use `WEBrick` on Linux to be consistent with OS X). Windows Git installation sadly doesn't provide the `git instaweb` command, but you can read how to set up gitweb using a separate web server such as Apache or IIS here: <https://git.wiki.kernel.org/index.php/MSysGit:GitWeb>.

Now that you have `git instaweb` set up, let's use it to display the repository in a browser.

Problem

You wish to display the contents of a repository in a browser.

Solution

- 1 Change to the directory containing your repository: for example,
`cd /Users/mike/GitInPracticeRedux/.`
- 2 Write a description for the repository's web server by running `echo "Git In Practice: Redux" > .git/description.`
- 3 Run `git instaweb --httpd=webrick`. There will be no output.

Git opens the gitweb interface in your browser; it should resemble figure 11.3. You can see that it displays a single Git project along with the description you just set, the owner, and the last change (commit) date. Click the Summary button to view more information about the `GitInPracticeRedux` project.

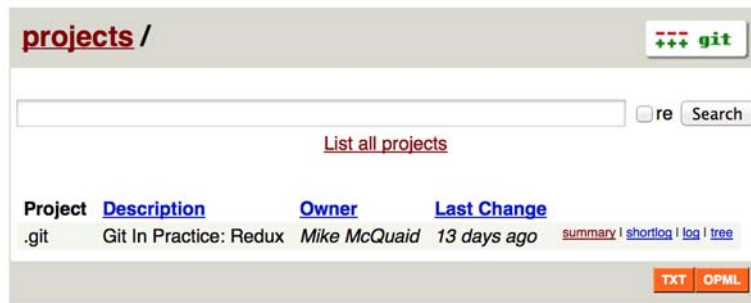


Figure 11.3
Gitweb projects

The summary page in figure 11.4 displays the same information as the projects page but also shows the list of recent commits, branches, and tags in a format resembling `GitX/gitk`.

Detailing all the features of the gitweb interface is beyond the scope of this book, but it's pretty self-explanatory. After you've finished exploring the gitweb interface, you can stop the server by running `git instaweb --stop`.

You have successfully displayed the contents of the repository in a browser.



Figure 11.4 Gitweb summary

Discussion

git instaweb can take some parameters to customize its behavior:

- `--local`—Ensures that the web server can only be accessed from the local machine and not from other machines on the same network.
- `--port`—Can be followed by a port number to specify which port should be used to access gitweb. For example, `--port 8080` means gitweb is hosted on port 8080.

11.1 Advanced Git hosting

In addition to the tools provided with Git that you've seen in this chapter, there is a wide third-party ecosystem of Git tools that can help you share your repositories and provide a web interface to view them. There are too many and their setup is too involved for me to detail them all here. Some of the most popular options are as follows:

- *GitHub* (<https://github.com>) is the most widely used Git host. It offers many features beyond sharing and viewing Git repositories. It provides free open source public hosting and paid private hosting. Alternatively, you can pay for GitHub Enterprise (<https://enterprise.github.com>), which provides a hosted GitHub appliance that can be run inside your network.
- *cgkit* (<https://github.com/zx2c4/cgkit>) provides a fast Git web interface written in C. It uses forking and a cache to speed up operations and is widely used by open source projects.
- *gitolite* (<https://github.com/sitaramc/gitolite>) provides access control for hosting Git repositories, such as users, groups, per-branch/per-repository permissions, and hook support.

11.2 Summary

In this chapter you learned the following:

- How to create a new bare repository with `git init --bare`
- How to mirror an existing repository with `git clone --mirror`
- How to share a repository across the network with `git daemon`
- How to display a web interface for a repository with `git instaweb`
- How to provide more advanced Git hosting with GitHub, cgit, and/or gitolite

Part 4

Git best practices

Part 4 (chapters 12–14) discusses different workflows and best practices for using Git as part of a team of software developers. You'll be able to compare and contrast differing approaches and decide which is best for your team.

This part will cover the following topics:

- How to write a good commit message and avoid whitespace problems
- How to build a commit using only certain changes in a file
- How CMake and Homebrew open source projects use differing merge/rebase strategies to manage contributions
- How to decide what merge/rebase strategy to use for your project
- How to decide what branching workflow to use for your project

12

Creating a clean history

This chapter covers

- Writing a good commit message
- Building a commit using only certain changes in a file
- Building a commit using a graphical application
- Avoiding committing bad whitespace

You saw in technique 4 that a clean history can make it easier to find useful information. For a history to be *clean*, it should use good, well-formatted commit messages for commits that are as small as possible. In this chapter, you'll learn how to create a clean history.

12.1 Writing a good commit message

Technique 4 explained why small commits are better and how commit messages should be formatted. Let's go into more detail about what makes a good format when writing commit messages, and why.

The following listing shows an example of a good commit-message format. It's strongly influenced by a guide written by Tim Pope, which is now at <http://mng.bz/hMe2>.

Listing 12.1 Good commit message format

Commits: first line is a summary (<51 characters).

Commit messages should be structured like emails. The first line (the commit subject) should be treated like the subject of an email. It should make a brief summary that is elaborated on in the rest of the commit message. It should have 50 characters or fewer and always be separated by a new line from the rest of the commit message body. Without this new line, various output formats that try to display only the first line of the commit may get confused.

The commit's message body can be split into multiple paragraphs which should be wrapped at 72 characters or fewer. The wrapping is done to ensure the output of tools like `git log` remains readable even when it adds indentation for diffs. Otherwise the commit message has no other limits on length; it should be as long as it needs to be to fully explain the commit. While the subject might describe what the commit does, the body should expand on why the change was made. It should also use the present tense to match the tense used by commit messages generated by commands such as `git merge`.

If you're using GitHub (and some other tools) then the contents of commit messages can contain Markdown. You may use Markdown to add some formatting that looks good in ASCII or rendered such as using **`**bold**`**, *`_italics_`*, ~~`~~strikethrough~~`~~, ``monospace`` or lists bulleted with a ``*`` or numbered with, for example ``1.``. You shouldn't go overboard with this but it can add some useful, basic formatting.

12.2 Building a commit from parts of files: `git add --patch`

You've seen previously in the book how to create commits from all the changes in an individual file. Because commits should be as small as possible (as discussed in technique 4), sometimes there may be multiple changes to a file that you want to split into multiple commits. Of course, you could manually undo and redo these changes to the files, but that would be tedious. Fortunately Git provides various tools you can use to add only certain changes in certain files to the index staging area or directly commit them.

`git add` has a `--patch` (or `-p`) flag that provides an interactive menu in which you can select what parts of files you want to add. Make some changes to `00-Preface.asciidoc`, `01-IntroducingGitInPractice.asciidoc`, and `02-AdvancedGitInPractice.asciidoc` in the `GitInPracticeRedux` repository, and run `git add --patch`. This will prompt for an action for the first change in `00-Preface.asciidoc` and should resemble the following.

Listing 12.2 Patch add output

```
# git add --patch

diff --git a/00-Preface.asciidoc b/00-Preface.asciidoc
index d7aa4f8..4b43488 100644
--- a/00-Preface.asciidoc
+++ b/00-Preface.asciidoc
@@ -1,2 +1,5 @@
```

```

= Git In Practice
+// Git Through Praxis?
The hotly anticipated sequel to Git In Paris.
+
+Copyright Mike McQuaid
Stage this hunk [y,n,q,a,d,/,e,?] ?
y - stage this hunk
n - do not stage this hunk
q - quit; do not stage this hunk nor any of the remaining ones
a - stage this hunk and all later hunks in the file
d - do not stage this hunk nor any of the later hunks in the file
g - select a hunk to go to
/ - search for a hunk matching the given regex
j - leave this hunk undecided, see next undecided hunk
J - leave this hunk undecided, see next hunk
k - leave this hunk undecided, see previous undecided hunk
K - leave this hunk undecided, see previous hunk
s - split the current hunk into smaller hunks
e - manually edit the current hunk
? - print help
...

```

← 1 Hunk line

← 2 Stage question

← 3 Help lines

- 1 shows the first line of the two-line *hunk*. A hunk consists of one or more nearby lines that have all been changed; Git groups them together.
- 2 shows the available options for this particular hunk. I selected `?`, which printed the shown help output.
- 3 shows the help output with all the different available options.

You can see from the help lines that many different options are available at each stage. These options should be fairly self-explanatory.

Let's use `s` to split this hunk into two hunks. After entering this, you'll see something resembling the following.

Listing 12.3 Output: *patch split*

```

Stage this hunk [y,n,q,a,d,/,s,e,]? s
Split into 2 hunks.
@@ -1,2 +1,3 @@
= Git In Practice
+// Git Through Praxis?
The hotly anticipated sequel to Git In Paris.

```

← 1 Split hunk

- 1 The previous hunk, which contained two changes (`// Git Through Praxis?` and `Copyright Mike McQuaid`), has been split into a single hunk. This lets you break hunks into smaller sections to allow the addition (and eventual committing) of individual lines.

After viewing the help, I answered `y` to stage the hunk and `q` to not stage any of the remaining hunks. If you do this, the `git status` output should resemble the following.

Listing 12.4 Output: status after patch add

```
# git status

On branch master
Your branch is up-to-date with 'origin/master'.

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   00-Preface.asciidoc          ← ❶ Staged file

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:   00-Preface.asciidoc
    modified:   01-IntroducingGitInPractice.asciidoc
    modified:   02-AdvancedGitInPractice.asciidoc      ❷ Unstaged file
```

❶ shows the file that had a hunk staged.

❷ shows the two files that have changes, none of which were staged, and the first file that had a single hunk staged and some hunks unstaged.

Undo this add to the staging area now by running `git reset master`.

`git commit` also has a `--patch` (or `-i` but, confusingly, not `-p`) flag. It provides the same interactive menu. Run `git commit --patch --message "Preface: add potential new title."`.

Listing 12.5 Output: patch commit

```
# git commit --patch --message "Preface: add potential new title."

diff --git a/00-Preface.asciidoc b/00-Preface.asciidoc
index d7aa4f8..4b43488 100644
--- a/00-Preface.asciidoc
+++ b/00-Preface.asciidoc
@@ -1,2 +1,5 @@
 = Git In Practice
+// Git Through Praxis?
+  The hotly anticipated sequel to Git In Paris.
+
+Copyright Mike McQuaid
Stage this hunk [y,n,q,a,d,,e,?]? s
Split into 2 hunks.
@@ -1,2 +1,3 @@
 = Git In Practice
+// Git Through Praxis?
+  The hotly anticipated sequel to Git In Paris.
Stage this hunk [y,n,q,a,d,,j,J,g,e,?]? y
@@ -2 +3,3 @@
+  The hotly anticipated sequel to Git In Paris.
+
+Copyright Mike McQuaid
Stage this hunk [y,n,q,a,d,,K,g,e,?]? q
```

```
[master eec78b2] Preface: add potential new title.    ← New commit
1 file changed, 1 insertion(+)
```

`git commit --patch` is equivalent to `git add --patch && git commit`. Perform the same actions with `git commit --patch` as with `git add --patch`: split the first hunk with `s`, stage the first hunk split with `y`, and then don't stage any of the others with `q`. The output is appended with the information you'd expect from `git commit` but is otherwise identical to that from `git add --patch`.

Now run `git reset HEAD^` to undo the current commit so you can try staging hunks graphically.

12.3 Graphically building a commit from parts of files

As you may have noticed, throughout this book I mostly prefer to use (and therefore teach you to use) the Git command-line application rather than GUIs. There are a few exceptions: I use GitX (or `gitk`, first seen in technique 4) to easily visualize the history of a repository.

You saw in section 12.2 how to build commits from parts of files from the Git command-line application, but it's a task I've found is far easier using a graphical application. In this section, I'll show you how to do this with GitX or Git Gui, which provides this functionality in a separate application.

12.3.1 Graphically building a commit in GitX

GitX provides a staging mode that lets you add entire files or individual hunks and provides support for graphically staging hunks. If you click the Stage button at upper right in GitX, it should resemble figure 12.1.

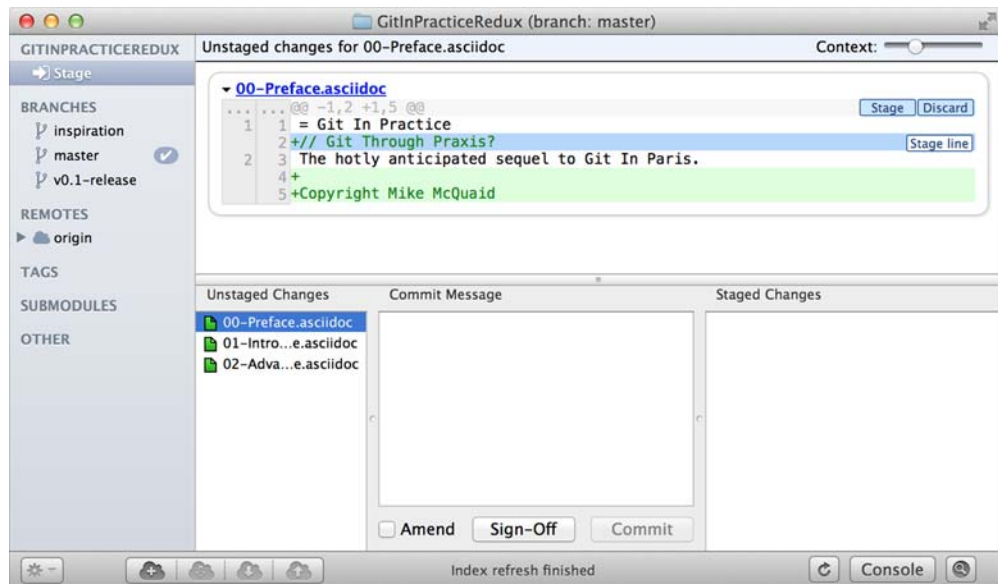


Figure 12.1 GitX stage mode

The staging mode shows a selection of files to stage and the changes to the selected file, and lets you stage hunks or selected lines through their respective buttons. Stage the `// Git Through Praxis?` line by clicking it and then clicking the Stage Line button that appears to the right of that line. If you want to stage all the lines in a hunk, you can click the Stage button at the top-right of the hunk. If you want to stage all changes in a file, you can right-click the filename in the Unstaged Changes list and select Stage Changes from the right-click menu.

Figure 12.2 shows GitX after the changes to the file were staged and a commit message was entered. The file now appears in both the Unstaged Changes and Staged Changes file lists. If the file had all its hunks staged, it would no longer appear in the Unstaged Changes list.

The staging area used by GitX is the same staging area used by the rest of Git. If you quit GitX now and ran `git status`, you'd see the same result as before: some changes in `00-Preface.asciidoc` were staged.

Now that there are some staged changes, the Commit button is enabled. After the commit message has been entered, you can click it.

Once the changes have been committed, you see a large message with the new SHA-1, as shown in figure 12.3. The Unstaged Changes list remains the same, but the items in the Staged Changes list were used to create the new commit, so they've been removed from this list.

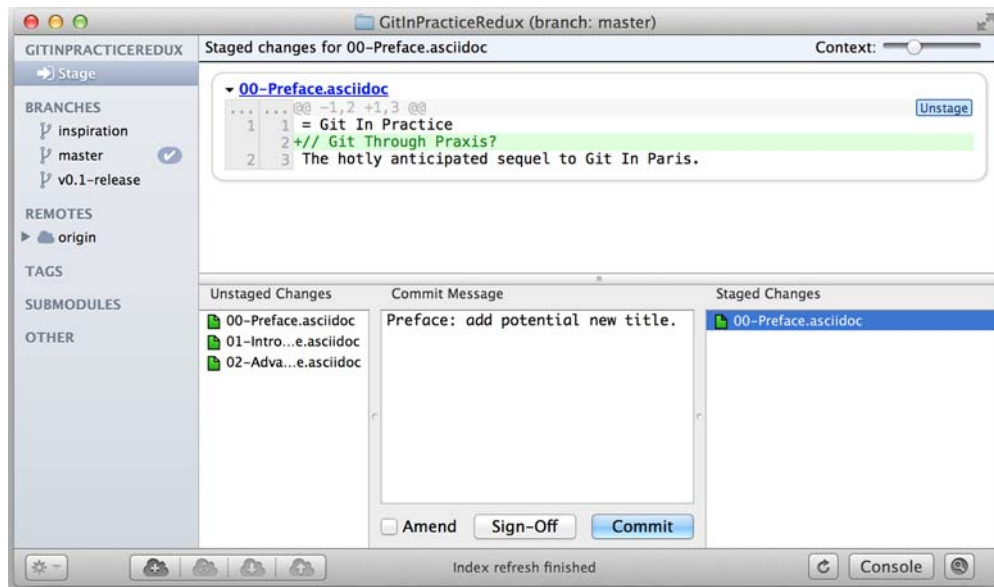


Figure 12.2 GitX staged hunk

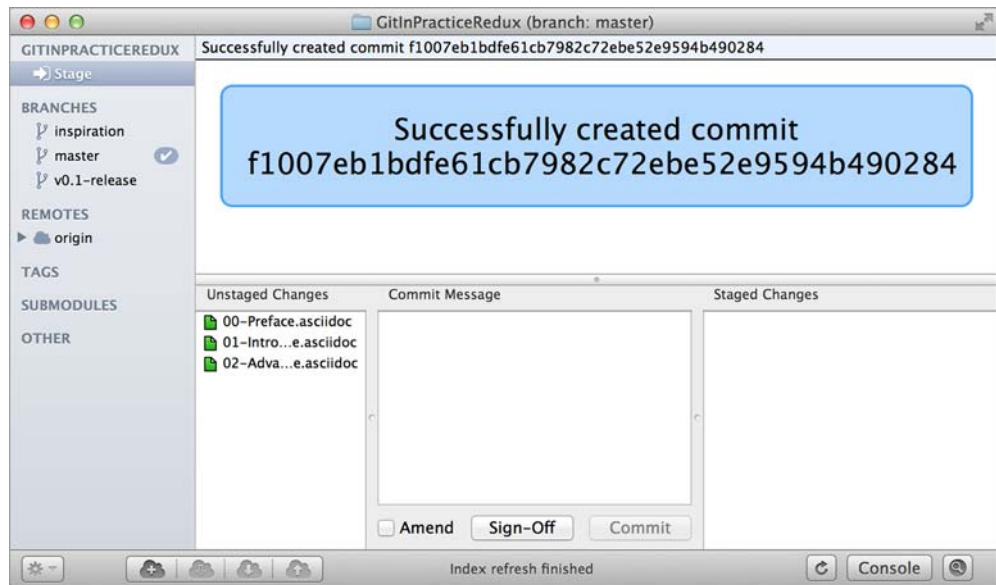


Figure 12.3 GitX stage-mode commit

12.3.2 Graphically building a commit in Git Gui

Although GitX combines staging and viewing history into one application, by default Git provides two GUI applications for this purpose: gitk (first seen in technique 4) and Git Gui. Run `git reset HEAD^` to undo the current commit; then run Git Gui.

Figure 12.4 shows the Git Gui user interface. It's similar to GitX's stage mode, but the Unstaged Changes and Staged Changes (Will Commit) file lists are shown on the left side rather than left and right of the commit message.

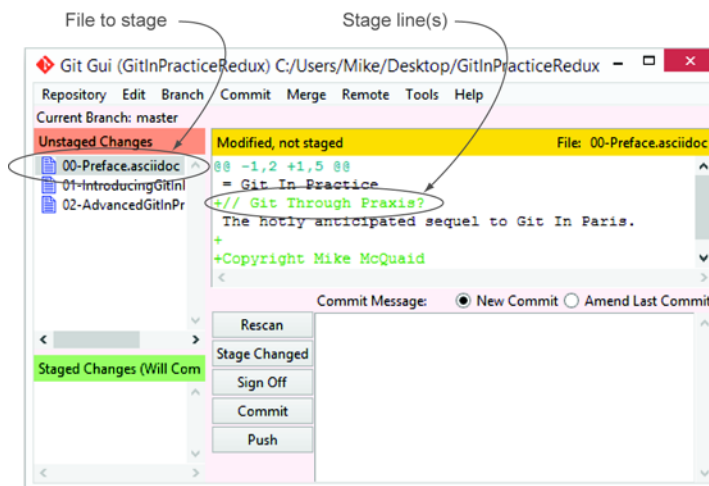


Figure 12.4 Git Gui on Windows 8.1

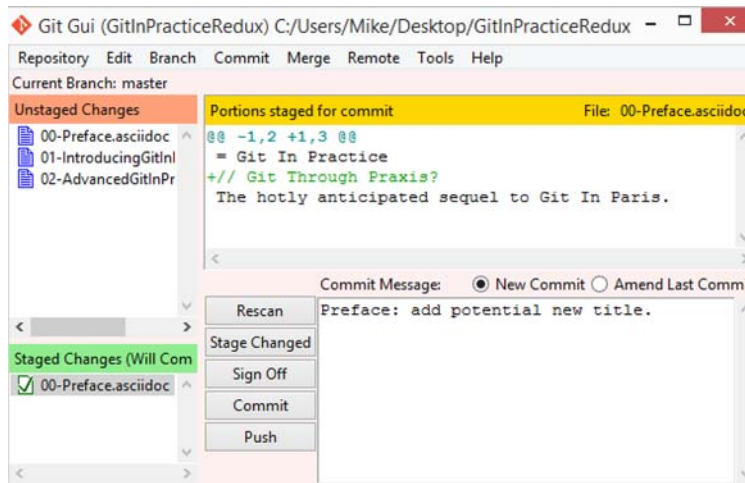


Figure 12.5 Git Gui staged line

You select the file whose changes you want to view by clicking it in the Unstaged Changes list. Stage the `// Git Through Praxis?` line by right-clicking it and selecting Stage Line for Commit from the right-click menu. If you want to stage all the lines in a hunk, you select Stage Hunk for Commit from the right-click menu. If you want to stage all changes in a file, select the filename in the Unstaged Changes list, click the Commit menu, and click Stage to Commit.

Figure 12.5 shows that a line has been staged in `00-Preface.asciidoc`: it's displayed in the Staged Changes (Will Commit) list. You can now enter a commit message and click Commit.

After you click Commit, there is no sign of the commit other than `00-Preface.asciidoc` being removed from the Staged Changes (Will Commit) list. Like GitX, though, Git Gui has successfully committed the file.

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

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

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

You can check that you haven't violated any of these rules by running `git diff --check`. For example, if you added some whitespace errors to `00-Preface.asciidoc`, the output might resemble the following.

Listing 12.6 Output: diff whitespace check

```
# git diff --check
00-Preface.asciidoc:1: trailing whitespace.      ← ❶ Trailing whitespace
+= Git In Practice
00-Preface.asciidoc:2: space before tab in indent. ← ❷ Space before tab
+      // Git Through Praxis?
```

❶ shows that line 1 of 00-Preface.asciidoc has whitespace at the end of the line.

❷ shows that line 2 of 00-Preface.asciidoc has a space character before a tab character at the beginning of the line.

Regular `git diff` (but, bizarrely, not `git diff --check`) shows \ No newline at end of file if the file's trailing newline is missing. If you have Git 2.0 (released May 28, 2014) or newer, or if you enabled colored output in section 7.1.1, `git diff` displays whitespace errors with a red background.

It's also worth checking whether you can configure your text editor of choice to fix any of these errors for you when you save files. This feature is fairly common.

12.5 Summary

In this chapter you learned the following:

- How to use an email format and Markdown to write good commit messages
- How to use `git add --patch` or `git commit --patch` to stage only chosen hunks for a new commit
- How to use GitX or Git Gui to stage only selected lines or hunks for a new commit
- How to use `git diff --check` to make sure you haven't added any bad whitespace changes

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.

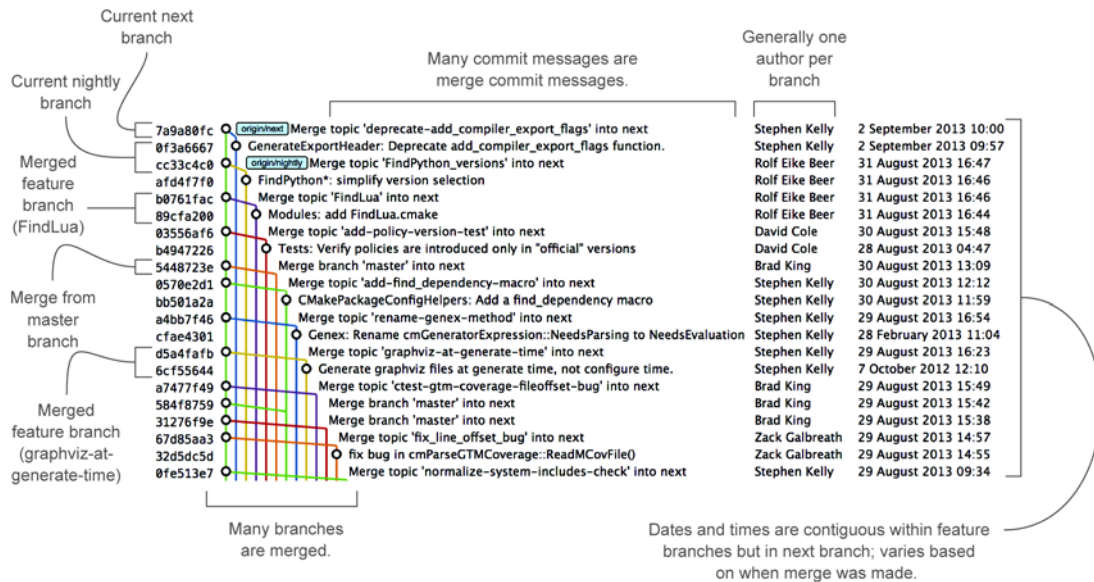


Figure 13.1 CMake repository history

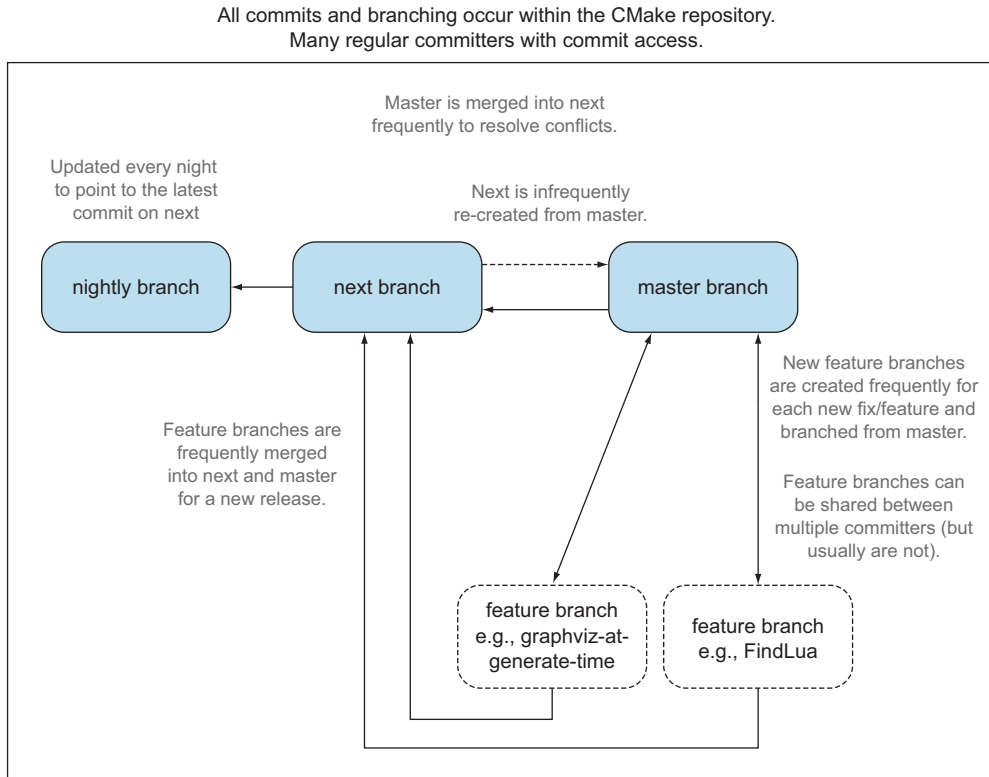


Figure 13.2 CMake branch/merge workflow

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.

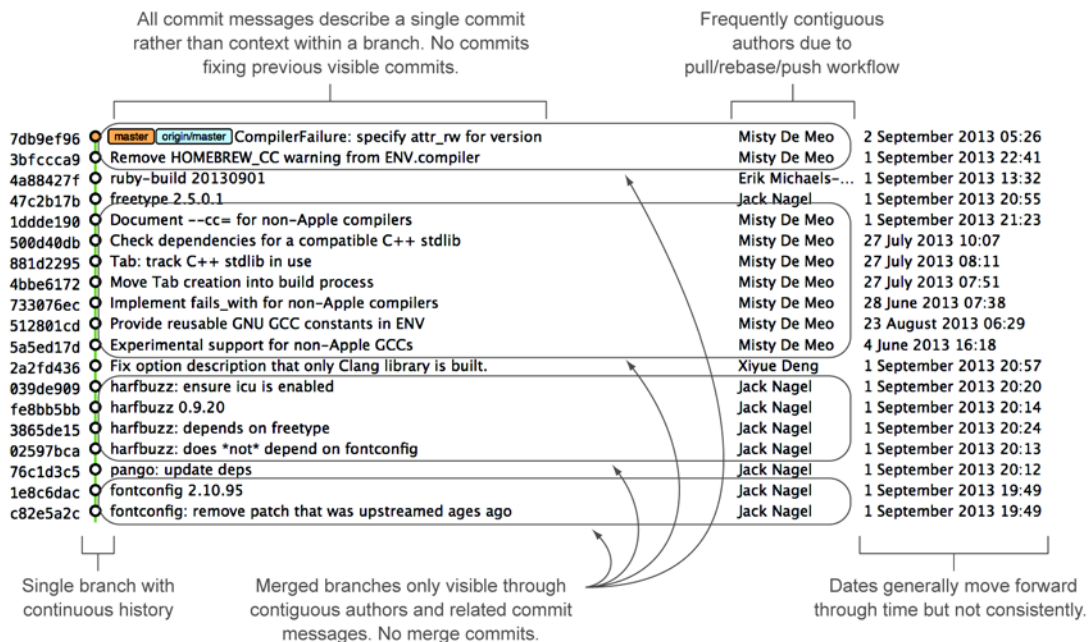


Figure 13.3 Homebrew repository history

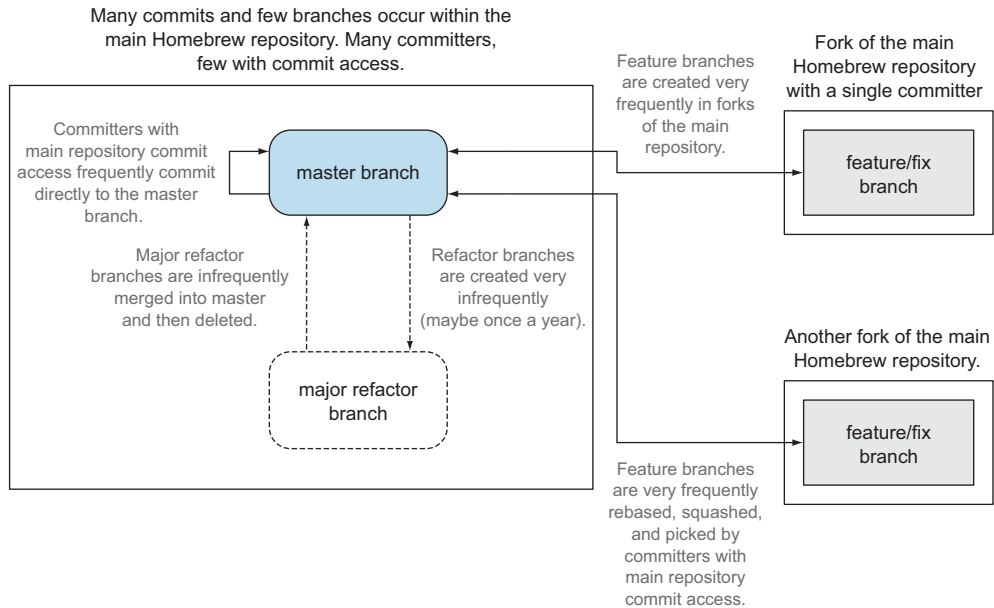


Figure 13.4 Homebrew's branch/rebase/squash workflow

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

14

Recommended team workflows

This chapter covers

- Using GitHub Flow to manage an untagged project
- Using Git Flow to manage a tagged project
- Using Mike Flow Single to manage a single-release project
- Using Mike Flow Multiple to manage a multiple-release project

You saw in sections 13.1 and 13.2 that two teams can both use Git as a version control system to manage a software project but handle branches, merging, and rebasing very differently. The different strategies for deciding how and when to branch, merge, or rebase as part of a team are called *team workflows*. These workflows allow for different ways of handling (or not handling) stable releases, tags, and bug fixing of previous releases. In this chapter, you'll learn about the two most popular Git workflows and two of my personal workflows.

14.1 GitHub Flow

GitHub Flow is so named because it's the Git workflow that came out of the GitHub organization. It was first formally described in a blog post by Scott Chacon in 2011 at <http://scottchacon.com/2011/08/31/github-flow.html>.

GitHub Flow is simple because it essentially involves only two types of branches: the default `master` branch and *feature branches*. A feature branch is one that is used only for the development of a single feature (or sometimes bug fix) and then deleted after being merged into another branch.

In GitHub Flow

- All commits are made on feature branches.
- Feature branches are merged to the `master` branch after review in a pull request.
- All commits to `master` are considered stable.

GitHub automatically deploys all commits to the `master` branch to the production web servers. You can read more about GitHub Flow on GitHub's guide at <https://guides.github.com/introduction/flow/index.html>.

Let's say you're working on a new web application called LintHub that lets people get together and share those annoying bits of fluff that accumulate in pockets. You want to add a feature for users to message each other, so you create a feature branch named `user-chat` from the `master` branch. You then commit your changes to this branch over the course of a few weeks and push them to the remote repository. When you've tested your changes and think they're stable enough to be used on the main web server, you create a pull request on GitHub. Your coworkers think your changes are OK, so you merge your changes to `master`, delete the `user-chat` branch, and deploy the changes to the main web server where they can be used by users. If you need to fix a bug with users being unable to message Scottish people, you can create another feature branch called `fix-scottish-users-chats` and follow the same process.

Let's look at figure 14.1 from top (the oldest commit) to bottom (the newest commit):

- 1 The initial commit to a repository is on the `master` branch and made with `git add` and `git commit`.
- 2 A new feature is being developed, so it's branched off the `master` branch with `git checkout -b`.
- 3 Another feature is developed in parallel, so it too is branched off the `master` branch with `git checkout -b`.
- 4 Commits are made to both feature branches with `git commit`. They're pushed periodically with `git push`. Commits may be rewritten locally with `git rebase --interactive` or `git commit --amend` before being pushed; but they're never rewritten after being pushed, so `git push --force` is never required.

- 5 A feature branch is submitted for review in a pull request either through the GitHub web interface or with `gh pull-request` (introduced in technique 59). If any changes are needed due to comments in the pull request, they're committed. The pull request is automatically updated after new commits have been pushed to the remote branch. When the branch is ready, it's merged to `master` with `git merge --no-ff` and deleted locally with `git branch --delete` and remotely with `git push --delete`.
- 6 The remaining feature branch needs to use the changes in `master` and so merges them with `git merge --no-ff`. This may be to use them in the branch or to resolve conflicts in the feature branch before they're merged into `master`.
- 7 The remaining feature branch is submitted for review, merged, and deleted.

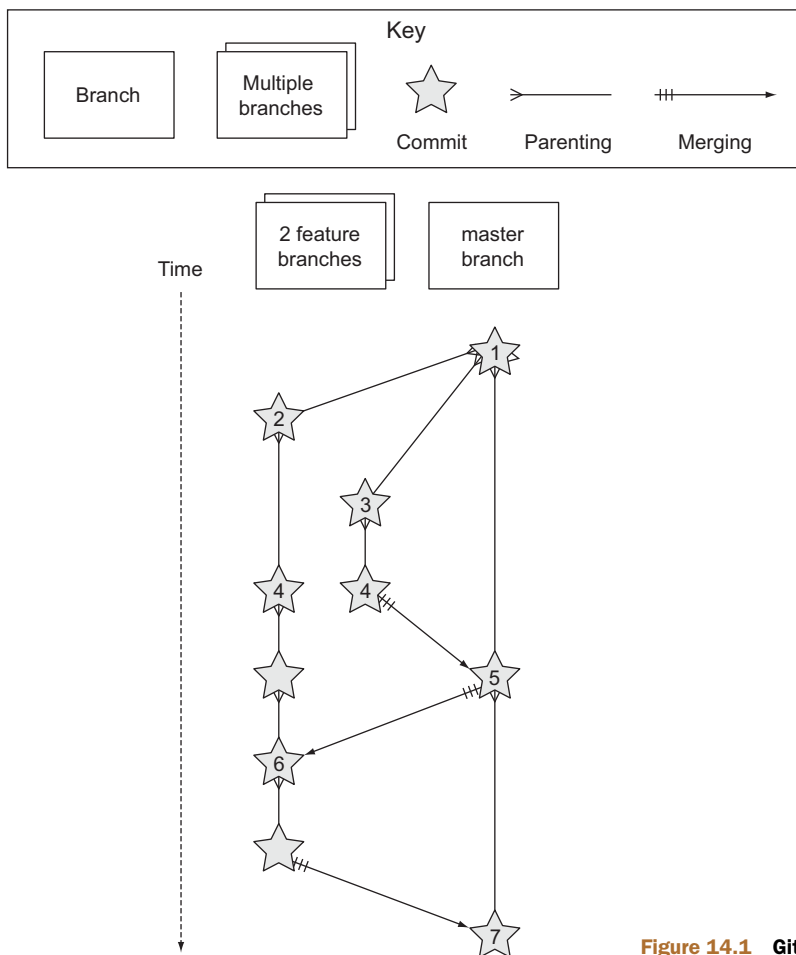


Figure 14.1 GitHub Flow

14.1.1 Pros and cons

The pros of GitHub Flow are as follows:

- It's beautiful in its simplicity. As a result, it's easy to use with Git, GitHub's web interface, and any Git graphical tools that support branching and merging.
- Because everything ends up in the `master` branch, there's little concern about commits getting lost.
- A branch is created, committed to, reviewed in a pull request, committed to again if necessary, merged, and deleted. If a branch exists, that's because either it hasn't been merged to `master` or someone forgot to delete it.

The cons of GitHub Flow are as follows:

- As a large web application, GitHub doesn't have versioned releases to customers. Every commit to the `master` branch is deployed to the production servers as soon as it's made. This approach is known as *continuous deployment*. It works well for web applications but doesn't work so well for things like desktop application software that needs to be released to users and requires them to restart their application to update it. In this case, because hotfixes can't be pushed to `master` without branching and review, it's important that feature branches be sufficiently tested before merging to `master`.
- Some pieces of software need to support multiple versions at once—for example, new stable versions of v1.0 and v2.0. GitHub Flow doesn't account for this case at all.

In my experience, GitHub Flow is a pleasantly simple workflow that's well suited for web projects that use continuous deployment. It's not well suited for desktop software, particularly when multiple releases need to be supported at once (release new v1.x and v2.x versions). These more advanced cases call for a more advanced workflow, such as Git Flow.

14.2 Git Flow

Git Flow was probably the first formal Git branching and merging workflow to take off. In 2010, Vincent Driessen wrote a blog post describing a Git workflow he'd been using for his software projects at <http://nvie.com/posts/a-successful-git-branching-model/>. The goals of Git Flow were to make heavy use of branching in Git and strict rules to provide a shared team understanding of the software release process while ensuring quality.

Git Flow involves five different types of branches:

- The `master` branch is used only for stable releases. Every commit to the `master` branch is tagged immediately. Commits are never made directly but are merged in from hotfix or release branches.

- The `develop` branch is a long-lived branch (it's never deleted). It's used for the bulk of daily development either through bug-fix commits directly to it or feature, release, or hotfix branch merges to it.
- *Feature branches* are branched off `develop` and are used for new feature development. When they're ready, they're merged back into `develop`. They can be named anything.
- *Release branches* are used to stabilize a release. They're branched off `develop` and have bug fixes made to them and hotfixes merged into them. Bug fixes and hotfixes are merged back into the `develop` branch. They're named prefixed with `release-`.
- *Hotfix branches* are used for small fixes that need an immediate new release. They're merged into `master` (where they're immediately tagged) and into the current release branch if it exists, or `develop` if it doesn't. They're named prefixed with `hotfix-`.

Let's say you're working on a new mobile application called GetPub that lets people get directions to the nearest pub. You want to add a feature for saving particular pubs, so you create a feature branch named `save-pubs` from the `develop` branch. You then commit your changes to this branch over the course of a few weeks and push them to the remote repository. When you've tested your changes and think they're stable enough to be included in the main development branch, you merge them into `develop` and delete your branch. When the new `2.x` release branch is created from `develop`, your Save Pub feature is included. After the release branch has been sufficiently tested, it's merged into `master`, tagged, and released to the users of the mobile application. If you need to fix a bug with saving Scottish pubs, you can create a hotfix branch named `hotfix-save-scottish-pubs` from the `2.x` release branch, commit to it, test it, and then merge it into `master`, tag it, and release it to your users.

A diagram will help you understand Git Flow. Let's look at figure 14.2 from top (the oldest commit) to bottom (the newest commit):

- 1 The initial commit to a repository is on the `develop` branch and made with `git add` and `git commit`.
- 2 A release branch for the `v1.x` release series is branched off the `develop` branch with `git checkout -b`.
- 3 A new feature is being developed, so it's branched off the `develop` branch with `git checkout -b`.
- 4 Commits are made to the `develop` branch for changes that aren't significant enough to be feature branches and aren't needed on the release branches. For example, small tweaks to behavior of the application that aren't fixes may be committed directly to the `develop` branch.
- 5 Another feature is developed in parallel, so it too is branched off the `develop` branch with `git checkout -b`.

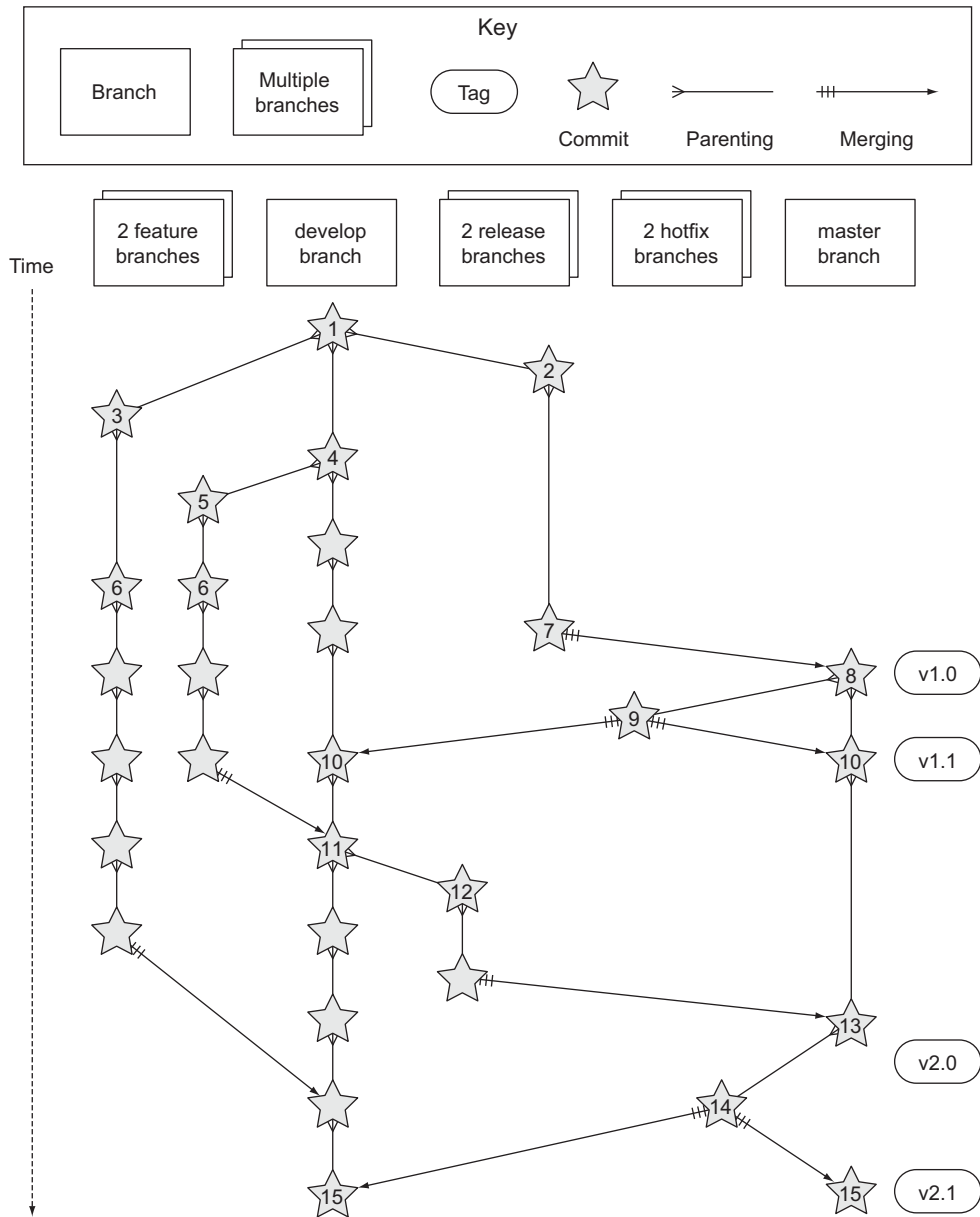


Figure 14.2 Git Flow

- 6 Commits are made to both feature branches with `git commit`. They may be pushed periodically with `git push` but typically to the developer's own remote repository rather than the main repository. Commits may be rewritten locally with `git rebase --interactive` or `git commit --amend` before being pushed but are never rewritten after being pushed, so `git push --force` is never required.

- 7 Commits are made to the release branches with `git commit` for changes or fixes specific to a release but not applicable to other application development. For example, such a commit may be changing a version number displayed in the application.
- 8 A release branch is ready and merged to master with `git merge --no-ff`. It's not deleted but is kept around indefinitely in case any more stable releases are needed from it. It's immediately tagged `v1.0` with `git tag` and pushed with `git push` and `git push --tags`.
- 9 A new hotfix is needed for both the develop branch and a release branch, so a hotfix branch is branched off the master branch with `git checkout -b`.
- 10 A hotfix branch is ready and merged to both develop and master with `git merge --no-ff`, and deleted locally with `git branch --delete` and remotely with `git push --delete`. The new master branch is immediately tagged `v1.1` with `git tag` and pushed with `git push` and `git push --tags`.
- 11 A feature branch is ready and merged to develop with `git merge --no-ff`, and deleted locally with `git branch --delete` and remotely with `git push --delete`.
- 12 A new release branch for the `v2.x` release series is branched off the develop branch with `git checkout -b`.
- 13 A release branch is ready and merged to master with `git merge --no-ff`. It isn't deleted but is kept around indefinitely in case any more stable releases are needed from it. It's immediately tagged `v2.0` with `git tag` and pushed with `git push` and `git push --tags`.
- 14 Another hotfix is needed for both the develop branch and a release branch, so another hotfix branch is branched off the master branch with `git checkout -b`.
- 15 A hotfix branch is ready and merged to both develop and master with `git merge --no-ff`, and deleted locally with `git branch --delete` and remotely with `git push --delete`. The new master branch is immediately tagged `v2.1` with `git tag` and pushed with `git push` and `git push --tags`.

Git Flow also has a set of Git extensions that let you work through this workflow with some commands. For example, to create a new feature branch, you can run `git flow feature start new-feature-name`. You can download these extensions and read more about them at <https://github.com/nvie/gitflow>.

14.2.1 *Pros and cons*

The pros of Git Flow are as follows:

- It allows you to keep track of released versions, features in development, and urgent and non-urgent bug fixes through branch naming.
- Having a formal flow through which branches are merged means a review process can ensure that things are reviewed multiple times before going into a release.

The cons of Git Flow are as follows:

- It's complicated to come to grips with. It can work well for organizations where people can be trained and onboarded, but it's less suitable for short projects or open source projects that seek to attract many new contributors.
- If you're using continuous deployment, the number of merges required from feature branch to master branch can be excessive.

In my experience, Git Flow is more complicated than it needs to be. Although it uses the flexibility of Git's branching, it has so many rules and restrictions that it becomes difficult to use. The previously mentioned extensions seem like a negative rather than positive for me; people clearly found Git Flow sufficiently difficult to work with that a separate layer had to be added on top of Git to use it effectively! It's definitely more complex to both use and understand than GitHub Flow, and it's harder to keep track of what changes are outstanding as a result. But as I mentioned earlier, GitHub Flow doesn't suit desktop application development as well as Git Flow. For these reasons, I've created my own Git workflow known as *Mike Flow*.

14.3 Mike Flow

I've used many different Git workflows over the years, from Git Flow, to GitHub Flow, to the Homebrew and CMake workflows mentioned in sections 13.1 and 13.2. They all have their pros and cons, but my preferred workflow is something I've named for this book: *Mike Flow*.

As you've seen with GitHub Flow and Git Flow, different release processes are optimized for continuously deployed web applications versus more slowly received desktop applications. For this reason, Mike Flow has two slightly different workflows: Mike Flow Single and Mike Flow Multiple.

The Single and Multiple in these cases refer to how many different versions of the software you need to support at once. Say you've made a v1.5 release and a v2.0 release. Will you make a v1.6 release after v2.0? If not, you want Mike Flow Single; and if so, you want Mike Flow Multiple. If you're doing continuous deployment, that's OK; Mike Flow Single can be used in this fashion too.

14.3.1 Mike Flow Single

Mike Flow Single is essentially GitHub Flow with two extra elements:

- 1 Branches can be (and should be) rebased, rewritten, and squashed where appropriate (to make history cleaner, but not if the branch is being used by multiple people).
- 2 Stable releases can be tagged on the master branch.

Let's say you're working on a new desktop application called GutRub that provides a tutorial for soothing digestion by rubbing your stomach. You want to add a feature for a counterclockwise rub tutorial, so you create a feature branch named counterclockwise-rub from the master branch. You then commit your changes to

this branch over the course of a few weeks and push them to the remote repository. When you've tested your changes and think they're stable enough to be included in the next version, you merge them into `master` and delete your branch. When version 2.0 is tagged and released from the `master` branch, it includes your Counterclockwise Rub feature. If you need to fix a bug with your tutorial playing backward, you can create a feature branch named `reverse-counterclockwise-rub` and follow the same process as with the new feature.

Let's look at figure 14.3 from top (the oldest commit) to bottom (the newest commit):

- 1 The initial commit to a repository is on the `master` branch and made with `git add` and `git commit`.
- 2 A new feature is being developed, so it's branched off the `master` branch with `git checkout -b`.

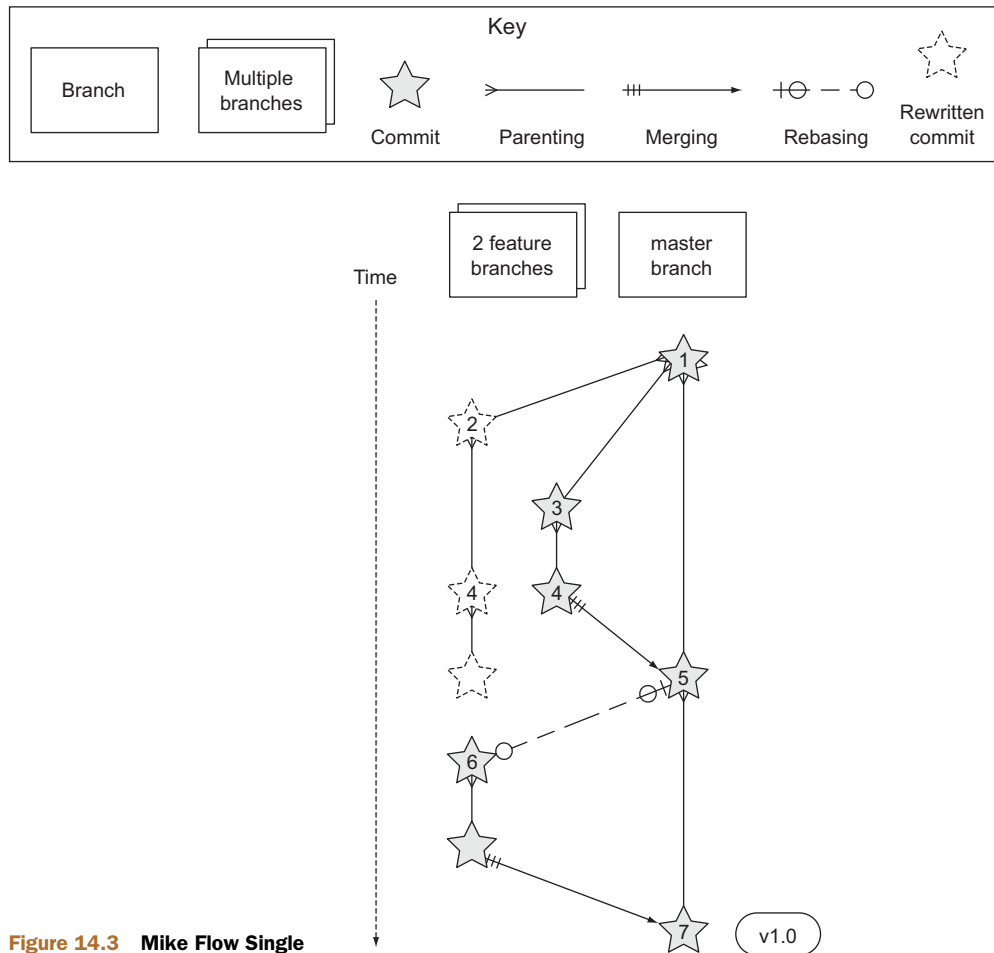


Figure 14.3 Mike Flow Single

- 3 Another feature is developed in parallel, so it too is branched off the master branch with `git checkout -b`.
- 4 Commits are made to both feature branches with `git commit`. They're pushed periodically with `git push`.
- 5 A feature branch is submitted for review in a pull request, either through the GitHub web interface or with `gh pull-request` (introduced in technique 59). If any changes to files are needed due to comments in the pull request, the changes are committed. The pull request is automatically updated with all changes after new commits have been pushed to the remote branch. When the branch is ready, it's merged to master with `git merge` and deleted locally with `git branch --delete` and remotely with `git push --delete`.
- 6 The remaining feature branch needs to use the changes in master and to rewrite commits, so it's rebased and squashed on top of master with `git rebase --interactive`. This may be to use work from master in the branch, to resolve conflicts in the feature branch before they're merged into master, or to clean up commits by rewriting them.
- 7 The remaining feature branch is submitted for review, merged, and deleted.

14.3.2 Mike Flow Multiple

Mike Flow Multiple is essentially Mike Flow Single with release branches:

- Release branches are branched off of master and can be committed to directly, cherry-picked, or merged to from feature branches.
- Unlike feature branches, release branches are never rewritten.
- Tags are created on feature branches rather than master.

Let's say you're working on a new desktop application called CutDub, which provides the ability to trim dubstep music albums to your chosen length. You want to add a feature for automatic silence trimming, so you create a feature branch named `auto-silence-trim` from the master branch. You then commit your changes to this branch over the course of a few weeks and push them to the remote repository. When you've tested your changes and think they're stable enough to be included in the next stable version, you merge them into master and delete your branch. When the `2.x` release branch is created from the master branch, it includes your automatic silence trimming feature. When `2.0` is tagged and released, it includes the feature. If you need to fix a bug in the `2.0` release with some trims causing corruption, you can create a feature branch named `fix-trim-corruption` and merge it into master and `2.x` and produce a new release after testing.

Figure 14.4 shows Mike Flow Multiple from top (the oldest commit) to bottom (the newest commit):

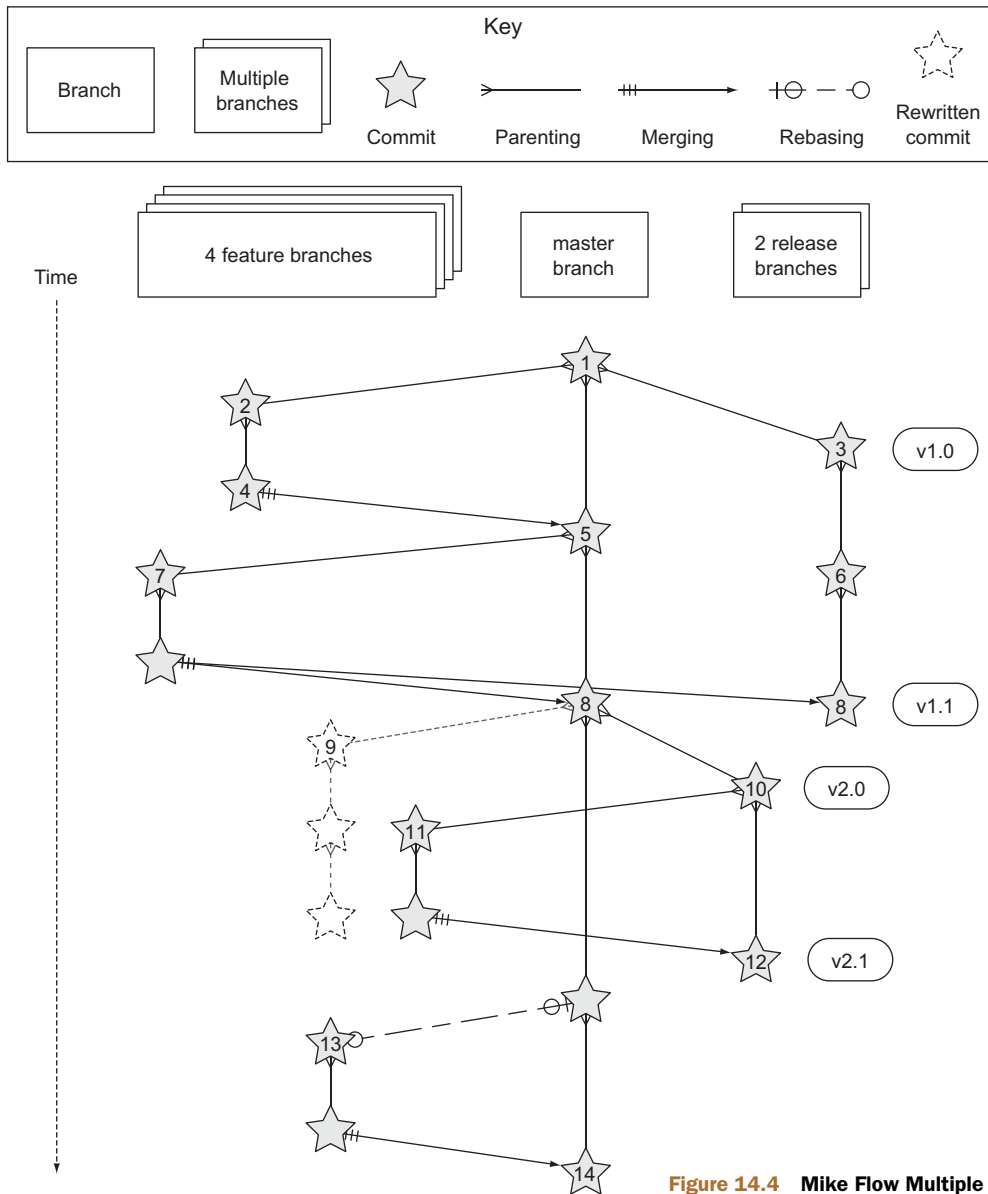


Figure 14.4 Mike Flow Multiple

- 1 The initial commit to a repository is on the master branch and made with `git add` and `git commit`.
- 2 A new feature is being developed, so it's branched off the master branch with `git checkout -b`.
- 3 A release branch for the v1.x release series is branched off the master branch with `git checkout -b`.

- 4 A commit is made to the feature branch with `git commit`. It's pushed with `git push`.
- 5 A feature branch is submitted for review in a pull request, either through the GitHub web interface or with `gh pull-request` (introduced in technique 59). If any changes are needed due to comments in the pull request, they're committed. The pull request is automatically updated after new commits have been pushed to the remote branch. When the branch is ready, it's merged to master with `git merge` and deleted locally with `git branch --delete` and remotely with `git push --delete`.
- 6 Commits are made to the release branches with `git commit` for changes or fixes specific to a release but not applicable to other application development. For example, such a commit may be changing a version number displayed in the application.
- 7 Another feature or fix is developed, so a new feature branch is branched off the master branch with `git checkout -b`.
- 8 A feature branch is ready and merged to both master and the release branch for the v1.x release series with `git merge` and deleted locally with `git branch --delete` and remotely with `git push --delete`. The new master branch is tagged v1.1 with `git tag` and pushed with `git push` and `git push --tags`. The release branch is not deleted but kept around indefinitely in case any more stable releases are needed from it.
- 9 Another feature is developed, so a new feature branch is branched off the master branch with `git checkout -b`.
- 10 A new release branch for the v2.x release series is branched off the master branch with `git checkout -b`.
- 11 A fix is needed for the v2.x release series, so a new feature branch is branched off the v2.x release branch with `git checkout -b`.
- 12 The feature branch fix is ready and merged to the v2.x release series branch with `git merge` and deleted locally with `git branch --delete` and remotely with `git push --delete`. The new master branch is tagged v2.1 with `git tag` and pushed with `git push` and `git push --tags`.
- 13 The remaining feature branch needs to use changes in master and to rewrite commits, so it's rebased and squashed on top of master with `git rebase --interactive`. This may be to use work from master in the branch, to resolve conflicts in the feature branch before they're merged into master, or to clean up commits by rewriting them.
- 14 The remaining feature branch is submitted for review, merged, and deleted.

14.3.3 Pros and cons

The pros of the two variants of Mike Flow are as follows:

- Any developers not interacting with a release can behave as if they're using GitHub Flow.
- Any developers who are more experienced with Git are empowered by being able to use more advanced history rewriting on remote branches. This allows them to keep their work shared and backed up but still make changes before it's merged.
- Tags and multiple-release branches are optionally added because they're necessary with some forms of software development, such as desktop applications where multiple versions need to be supported.

The cons of the two variants of Mike Flow are as follows:

- It's not a known workflow outside of this book and people who have worked with me! That said, I've known many other projects that have adopted a similar process.
- Its flexibility in history rewriting and branching may lead to more mistakes.

Obviously I'm biased, but I think Mike Flow provides the best of both Git Flow and GitHub Flow: a stable release and review structure but without the complexity of a strictly mandated process.

14.4 Which workflow is for you?

It's worth reading about and trying to understand all the workflows in this chapter and in sections 13.1 and 13.2 before deciding on a workflow for your own team. Here are some questions to ask yourself when picking the best workflow:

- Is your team experienced with Git or new to it? If new to it, you probably want to pick as simple a workflow as possible (not Git Flow).
- Does your team respond better to a rigid, documented process with strict rules or a more relaxed approach? If they like a rigid process, then Git Flow may be a good solution.
- Do you want to do multiple reviews before releasing code to customers? If so, Git Flow may be a good fit.
- Do you need to release new versions for multiple release series? For example, after releasing v2.0 will you later release v1.5? If so, Mike Flow Multiple or Git Flow will be a good bet. GitHub Flow doesn't handle this situation at all, so be extremely wary if you're considering using it in this case.
- If you want to use workflows that are well-known outside this book, it's best to stick to GitHub Flow or Git Flow.
- If you want to deploy code to production as quickly as possible, it's perhaps worth sticking with GitHub Flow or Mike Flow Single, because Git Flow and Mike Flow Multiple add intermediate steps.

- If you're working alone but still want to use a structured branching workflow, GitHub Flow and Mike Flow Single and Multiple are sufficient, and Git Flow is overkill.

Ultimately, any of the workflows discussed in this book will be better than no workflow at all (but do try to be consistent). Remember that Git is a powerful tool, and it should help you and your team be more productive and write better software. Good luck with it!

14.5 Summary

In this chapter you learned the following:

- How to use GitHub Flow to use feature branches and master for continuous deployment
- How to use Git Flow to create a strict release, bug fix, and feature development process
- How to use Mike Flow to create a simple workflow for non-release operations with the power of history rewriting

appendix A

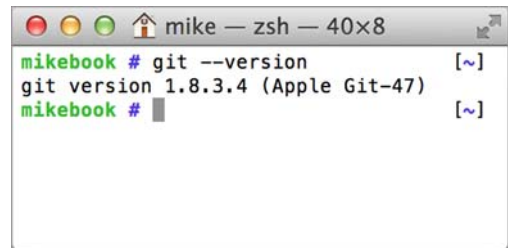
Git installation

Let's see if Git is already installed on your local machine and install it if needed. Git doesn't come preinstalled on many operating systems, because it's a tool typically used by programmers rather than nontechnical computer users. The method required to install Git varies depending on your operating system of choice.

WHY ARE THERE DIFFERENT VERSIONS OF GIT IN THIS APPENDIX? The different installation methods and operating systems install different versions of Git. Don't worry about this; the main differences between newer Git versions and older ones are the helpfulness of the output messages. Version 2.0.0 was the released version at the time of writing, but version 1.8 or above should be sufficient for the needs of this book.

A.1 How to install and run Git on Apple OS X

To verify whether Git is already installed, open a Terminal (either the default OS X /Applications/Utilities/ Terminal.app or an alternative such as iTerm.app) and run `git --version`. If Git is already installed, the output should resemble figure A.1.



```
mikebook # git --version [~]
git version 1.8.3.4 (Apple Git-47)
mikebook # [~]
```

Figure A.1 `git --version` in Terminal.app on OS X Mavericks

A.1.1 Installing Git on Apple OS X Mavericks or newer

If you're running OS X Mavericks (10.9) or newer and Git wasn't already installed when you ran `git --version`, it will prompt you to download and install Git similarly to figure A.2.

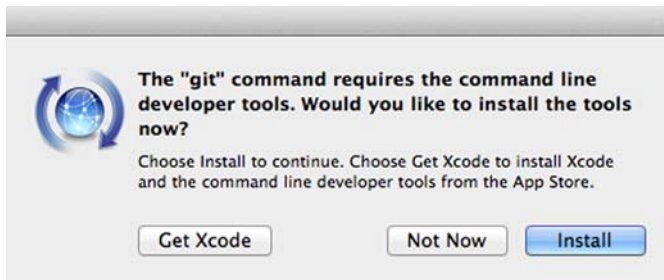


Figure A.2 OS X Mavericks Git installation

A.1.2 Installing Git on Apple OS X Mountain Lion or older

If you're running OS X Mountain Lion (10.8) or older and you have a package manager installed, you can install Git using one of the following options:

- *Homebrew/Tigerbrew (recommended)*—`brew install git`
- *MacPorts*—`sudo port install git-core +svn`
- *Fink*—`fink install git`

If you don't wish to install or use a package manager, you can install Git using a graphical installer from the official Git site at <http://git-scm.com/download/mac>.

A.2 How to install and run Git on Linux or Unix

To verify whether Git is already installed, open a Terminal application or console and run `git --version`. If Git is already installed, the output should resemble figure A.3.

On Linux or Unix, you can install Git directly from your package manager. How to do this varies from system to system, but here are some of the popular options:

- *Debian/Ubuntu*—`apt-get install git`
- *Fedora*—`yum install git`
- *Gentoo*—`emerge --ask --verbose dev-vcs/git`
- *Arch Linux*—`pacman -S git`
- *FreeBSD*—`cd /usr/ports/devel/git && make install`
- *Solaris 11 Express*—`pkg install developer/versioning/git`
- *OpenBSD*—`pkg_add git`



Figure A.3 `git --version` in XFCE Terminal on Debian 7.2 (Wheezy)

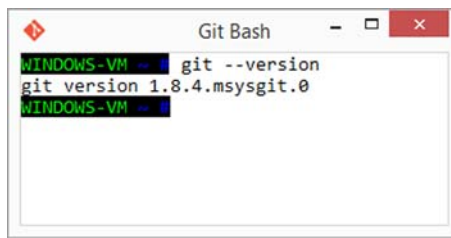
A.3 How to install and run Git on Microsoft Windows

To verify whether Git is already installed, look for Git Bash links in your Start menu or on your Desktop. Git for Windows can be downloaded from the official Git site at <http://git-scm.com/download/win>. Download and click through the installer. When it has completed, it will provide Start menu links to run Git Bash.

Because Git is a Unix program, running Git on Windows runs a Unix shell that allows access to Git commands. This may be slightly scary, but don't worry; this book will show you any commands you need to use.

To run Git commands, open the Git Bash shortcut from the Start menu. This will open a Unix shell in a Windows Command Prompt.

With the Git shell open, you can type in Git commands. To see what Git version you have installed, type `git --version`. The output should resemble figure A.4.

A screenshot of a Git Bash terminal window. The window title is "Git Bash". The prompt is "WINDOWS-VM ~". The command entered is "git --version". The output is "git version 1.8.4.msysgit.0". The prompt is now "WINDOWS-VM ~".

```
Git Bash
WINDOWS-VM ~$ git --version
git version 1.8.4.msysgit.0
WINDOWS-VM ~$
```

Figure A.4 `git --version` in Git Bash on Windows 8.1

A.4 Verifying that Git has installed correctly

To run Git commands, you need to open a Terminal application, console, or command prompt (depending on your platform). To verify that Git has installed correctly, run `git --version`, which should output `git version 1.8.5.2` (or another version).

appendix B

Creating a GitHub account and repository

GitHub is a website that provides Git repository hosting as well as issue trackers, Git-backed wikis, and a workflow to request a merge of the commits in a branch (which is known as a *pull request* and is discussed in section 10.1). You can create free accounts for public remote repositories, where everyone can see your code and commits. Typically these are used by open source projects, but they will also prove useful for your learning and experimentation.

As mentioned in chapter 2, there are free and paid alternatives to GitHub. I've picked GitHub to walk through because, at the time of writing, it's the most popular hosted version control system for open source projects and is probably the most popular Git hosting provider. Learning to use GitHub will bring you immediate benefits in terms of facilitating open source access and contributions. Although the GitHub UI may differ from the examples here or from other Git repository hosts, the Git commands used will be the same.

B.1 Signing up for a GitHub account

Let's sign up for a new GitHub account. Browse to <https://github.com/join>, where you should see something like figure B.1.

This form allows you to create a new GitHub account, which will let you access the service and create new repositories. The username you pick will determine the URL of your GitHub account page and be part of the URL for every repository you create, so choose it carefully. It can be renamed in the future, but this may cause problems when updating existing local repositories without manually changing the URL.

The screenshot shows the GitHub 'Join GitHub' form. At the top, it says 'Join GitHub' and 'The best way to design, build, and ship software.' Below this are three steps: Step 1: Set up a personal account (active), Step 2: Choose your plan, and Step 3: Go to your dashboard. The main section is 'Create your personal account'. It has four input fields: a username field with 'GitInPractice', an email field with 'gitinpractice@mikemcquaid.com', a password field with '*****', and a confirmation password field with '*****'. Each field has a green checkmark to its right. To the right of the form is a box titled 'You'll love GitHub' with the following text: 'Unlimited collaborators', 'Unlimited public repositories', 'Great communication', 'Friction-less development', and 'Open source community'. At the bottom of the form is a green button labeled 'Create an account'. Four numbered annotations are present: 1. 'Enter a username.' with an arrow pointing to the username field. 2. 'Enter your email.' with an arrow pointing to the email field. 3. 'Enter a password.' with an arrow pointing to the password field. 4. 'Click Create an Account.' with an arrow pointing to the 'Create an account' button.

1 Enter a username.

2 Enter your email.

3 Enter a password.

4 Click Create an Account.

Figure B.1 Join GitHub form

Enter your username, email, and password, and click the Create button to advance to the next screen (see figure B.2). This form allows you to select your GitHub payment plan. The only differences between plans are the number of private repositories you can create. A *private repository* means none of your commits or files committed to the repository can be accessed by others without your explicit approval. A *public repository* means all your commits are viewable by anyone but nobody else can commit to your repository without your explicit approval. In this book, you never have to commit anything private to a repository, so you don't need to choose a paid plan. After you've selected a plan, click the Finish button to advance to the next screen.

You have created a GitHub account, and the next step is to create a new repository.

Welcome to GitHub

You've taken your first step into a larger world, @GitInPractice.

Completed
Set up a personal account

Step 2:
Choose your plan

Step 3:
Go to your dashboard

Choose your personal plan

Plan	Cost	Private repos	
Large	\$50/month	50	Choose
Medium	\$22/month	20	Choose
Small	\$12/month	10	Choose
Micro	\$7/month	5	Choose
Free	\$0/month	0	Choose

Each plan includes:

- Unlimited collaborators
- Unlimited public repositories
- ✓ Free setup
- ✓ SSL Protection
- ✓ Email support
- ✓ Wikis, Issues, Pages, & more

Don't worry, you can cancel or upgrade at any time.

☐ **Help me set up an organization next**
Organizations are separate from personal accounts and are best suited for businesses who need to manage permissions for many employees.
[Learn more about organizations.](#)

Finish sign up

1 Optional: choose a paid, private plan.

2 Click Finish Sign Up.

Figure B.2 Choosing a GitHub plan

B.2 Creating a new repository on GitHub

After signing up for your new GitHub account, you should see your dashboard, which should resemble figure B.3. On the dashboard are buttons you can click to create a new GitHub repository. Click to advance to the next screen.

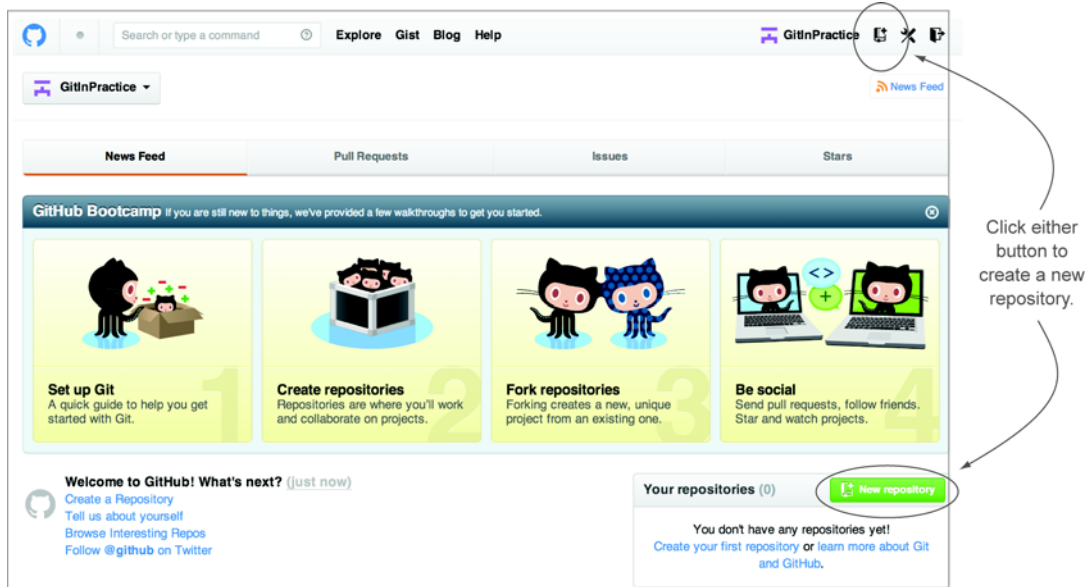


Figure B.3 Dashboard buttons to create a new GitHub repository

Creating a new repository requires you to pick a name and, optionally, a description, as shown in figure B.4. This name will be combined with the username you chose earlier to make the URL for your repository, so choose it carefully. It can be renamed in the future, but this may cause problems when updating existing local repositories without manually changing the URL. You may also choose for the repository to be private, which requires purchasing a paid GitHub plan. After entering the repository details, click the Create button to advance to the next screen.

1 Enter repository name

2 Optional: enter a repository description.

3 Optional: choose a private repository (and a paid plan).

4 Click Create Repository.

The form shows the 'Owner' as 'GitInPractice' and the 'Repository name' as 'GitInPractice'. The 'Description (optional)' is 'Git In Practice'. The 'Public' radio button is selected. The 'Initialize this repository with a README' checkbox is unchecked. The 'Add .gitignore' dropdown is set to 'None' and the 'Add a license' dropdown is also set to 'None'. A green 'Create repository' button is at the bottom.

Figure B.4 Creating a new GitHub repository

You have created a GitHub repository and should see something similar to figure B.5.

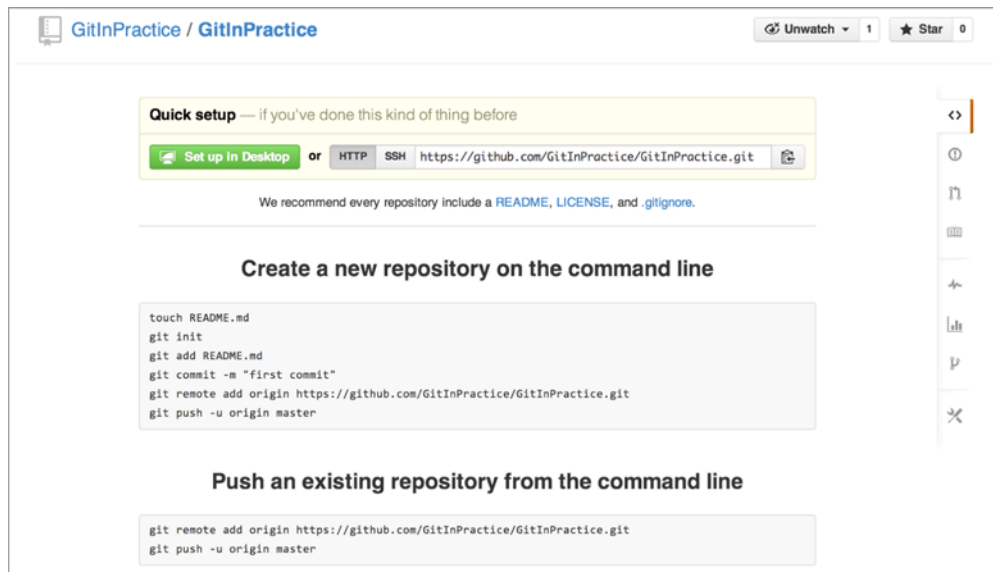


Figure B.5 A new GitHub repository

appendix C

Commented Git configuration

In chapter 7, you learned about configuring Git for maximum productivity. I've been using Git as my main version control system for a pretty long time at this point, so I have my Git configuration well optimized.

This appendix includes my Git configuration files at the time of writing. They're heavily commented to explain what they're doing and why. The latest versions are always available in my dotfiles repository on GitHub: <https://github.com/mikemcquaid/dotfiles>

Listing C.1 `~/.gitconfig`

```
[include]
  path = ~/.gitconfig-user
[color]
  # Colour terminal command output when possible.
  ui = auto
[gist]
  # Open GitHub Gist in browser after submission.
  browse = true
[push]
  # Push to the set upstream branch being tracked by default.
  default = simple
[fetch]
  # Always prune when fetching (and pulling).
  prune = 1
[rerere]
  # Store and re-use manual conflict resolution changes.
  enabled = 1
[core]
  # Exclude everything this file. Used for general exclusions.
  excludesfile = ~/.gitignore
  # Set attributes on files. Used for general diff improvements.
  attributesfile = ~/.gitattributes
[help]
  # Open Git help pages in the browser as HTML.
```

```

format = web
autocorrect = 1
[alias]
## 'New' Commands
# Stage the changes in a given file.
stage = !git add -u
# Unstage the changes in a given file.
unstage = reset HEAD --
# View the current changes in the staging area.
staged = diff --cached
# Print the name of the current branch.
current-branch = symbolic-ref --short HEAD
# Print the name of the current upstream tracking branch.
upstream = !git config --get branch.$(git current-branch).remote \
    || echo origin
# Cherry-pick a commit with your signature.
sign = cherry-pick --signoff
# List all current SVN externals for this repository.
svn-externals = !git svn show-externals | grep -x \"^[^#].*\"
# Create a git:// server of the current repository.
# WARNING: this gives all users read/write access
# without authentication (so only use on trusted networks).
serve = !git daemon --reuseaddr --export-all --base-path=. \
    --verbose ./git
# Merge a branch and commit a merge commit (even if one
# isn't needed)
noff = merge --no-ff
# Merge a branch with a merge commit and resolve any conflicts
# always using that branch's version rather than the current branch.
theirs = !git noff -Xtheirs
# Fetch all branches and rebase the current branch against
# upstream/master.
rebase-against-master = !git fetch --all \
    && git rebase $(git upstream)/master
# Push the current branch upstream to origin using the same branch
# name for the remote branch.
upstream-current-branch = !git push --set-upstream origin \
    $(git current-branch)
# Create a pull request on GitHub using the `gh` command.
pull-request = !git upstream-current-branch && gh pull-request
# Upstream the current branch to origin and create a pull request
# on GitHub.
upstream-and-pull-request = !git upstream-current-branch \
    && git pull-request
# Get the current diff but show differences between characters
# instead of just the differences between lines.
word-diff = diff --word-diff
# Push the current branch and set it as the default upstream branch.
push-and-set-upstream = push --set-upstream
# Create a new branch by checking out another branch.
checkout-as-new-branch = checkout -b
# Rebase against origin/master and prompt for what operations
# should be performed.
interactively-rebase-against-origin-master = \
    !git rebase --interactive origin/master

```

```

# Show the commit log with a prettier, clearer history.
pretty-one-line-log = log --graph --oneline --decorate
# Commit any changes to files, squash them into the last commit
# and update its date.
fix-up-previous-commit = !git commit --all --amend \
                        --reuse-message=HEAD --date="\$(date)\\" #"
# Checkout the staging branch and update it.
pull-staging = !git checkout staging && git pull
# Checkout the master branch and update it.
pull-master = !git checkout master && git pull
# Update the staging, master and current branches.
pull-master-staging-and-current-branch = \
    !BRANCH=$(git current-branch) && git fetch --all \
    && git pull-staging && git pull-master && \
    git checkout $BRANCH && git pull
# Commit a work-in-progress commit (to use with
# fix-up-previous-commit)
work-in-progress = commit -a -m 'WIP'
# Merge a branch with a merge commit and use the more time-consuming
# patience diff algorithm
patience = !git noff -Xpatience
# Hard reset branch to the upstream version.
hard-reset = !git reset --hard $(git upstream)/$(git current-branch)
# Assume the specified file is unchanged to stop changes
# being seen by Git
assume = update-index --assume-unchanged
# No longer assume a specified file remains unchanged
unassume = update-index --no-assume-unchanged
# List all files that are assumed to be unchanged
assumed = !git ls-files -v | grep '^[hsmrck?]' | cut -c 3-

## Shortened 'New' Commands
fahr = !git fetch --all && git hard-reset
rem = !git rebase-against-master
wip = !git work-in-progress
pr = !git upstream-and-pull-request
up = !git upstream-current-branch
pm = !git pull-master-staging-and-current-branch
fa = !git fetch --all
w = !git word-diff
u = !git push-and-set-upstream
b = !git checkout-as-new-branch
i = !git interactively-rebase-against-origin-master
# `true` needed as the return status is wrong otherwise.
l = !git pretty-one-line-log || true
f = !git fix-up-previous-commit

## Shortened Existing Commands
p = pull
s = status --short --branch

[instaweb]
# Use the Ruby WEBRick library when creating a `git instaweb`
# HTTP server.
httpd = webrick

[diff]
# Use the OS X graphical three-way merge tool for graphical diffs.

```

```

    tool = opendiff
    # Use the slower but better patience diff algorithm
    algorithm = patience
[diff "xml"]
    textconv = xmllint --format --recover
[difftool "opendiff"]
    # Set the OS X opendiff command name.
    path = opendiff
[merge]
    # Use the OS X graphical three-way merge tool for graphical merges.
    tool = opendiff
[mergetool]
    # Don't prompt before opening the merge tool.
    prompt = false
    # Don't keep backups of the merge tool inputs.
    keepBackup = false
    # Don't keep the merge tool temporary input/output files.
    keepTemporaries = false
[mergetool "opendiff"]
    # Use a script to setup opendiff correctly for Git merges.
    path = git-mergetool-opendiff
[apply]
    # Cleanup whitespace by default when apply patches.
    whitespace = fix
[url "git@github.com:"]
    # Always use GitHub SSH protocol to push.
    # Allows use of git:// for public repositories with push access
    pushInsteadOf = git://github.com/
[url "https://github.com/"]
    # Use HTTP for GitHub instead of git:// or git@
    # Enable this in networks where git:// or git@ are blocked.
    #insteadOf = git://github.com/
    #insteadOf = git@github.com:
[url "git@github.com:"]
    # Use SSH for GitHub instead of https://
    # Enable this in networks where https:// has issues.
    #insteadOf = https://github.com/
[url "git@gitorious.org:"]
    # Always use Gitorious SSH protocol to push.
    # Allows use of git:// for public repositories with push access
    # (which is often faster).
    pushInsteadOf = git://gitorious.org/
[credential]
    # Use OSX Keychain to store HTTP passwords.
    helper = osxkeychain
[filter "media"]
    clean = git-media-clean %f
    smudge = git-media-smudge %f
[gh]
    protocol = https

```

Listing C.2 ~/.gitconfig-user

```

[user]
    # Name used in commit messages.

```

```

    name = Mike McQuaid
    # Email used in commit messages.
    email = mike@mikemcquaid.com
[shell]
    # Default SSH username.
    username = mike
[sourceforge]
    # SourceForge username.
    username = mikemcquaid
[github]
    # GitHub username for command-line tools.
    user = mikemcquaid
[alias]
    # Push the current branch upstream to mikemcquaid using the same
    # branch name for the remote branch.
    um = !git push --set-upstream mikemcquaid $(git current-branch)

```

Listing C.3 `~/.gitignore`

```

# Ignore files generated by Qt.
moc_*.cpp
qrc_*.cpp
ui_*.h

# Ignore compiler output files.
*.o
*.pyc

# Ignore text editor local configuration..
*.pro.user
.tm_properties
*.xcodeproj/project.xcworkspace/
*.xcodeproj/xcuserdata/

# Ignore temporary generated files.
*.rej
*.swp
*~

# Ignore thumbnails metadata generated by OSX.
.DS_Store

# Ignore wrapper scripts generated by Bundler.
.bundle/bin/

# Ignore files generated by CMake.
CMakeFiles/
CMakeCache.txt
cmake_install.cmake
install_manifest.txt

# Ignore build directories.
b/

```

Listing C.4 `~/.gitattributes`

```

# Diff .t2d files as if they were XML (they are).
*.t2d diff=xml

```


appendix D

Why use version control?

You may sometimes find yourself speaking to people who aren't familiar with version control concepts or why version control systems are useful for managing changes to text. This appendix provides a simple but effective argument for convincing others why version control is useful and important.

D.1 Handling changes

A common problem when dealing with information stored on a computer is handling *changes*. For example, after adding, modifying, or deleting text, you may want to undo that action (and perhaps redo it later). At the simplest level, this might be done by clicking Undo in a text editor (which reverts a previous action); after new words are added, it may be necessary to undo these changes by pressing Undo repeatedly until you return to the desired previous state.

A naïve method for handling multiple file versions is often to create duplicate files with different filenames and contents (Important Document V4 FINAL FINAL.doc may sound sadly familiar). An example of this approach can be seen in figure D.1.

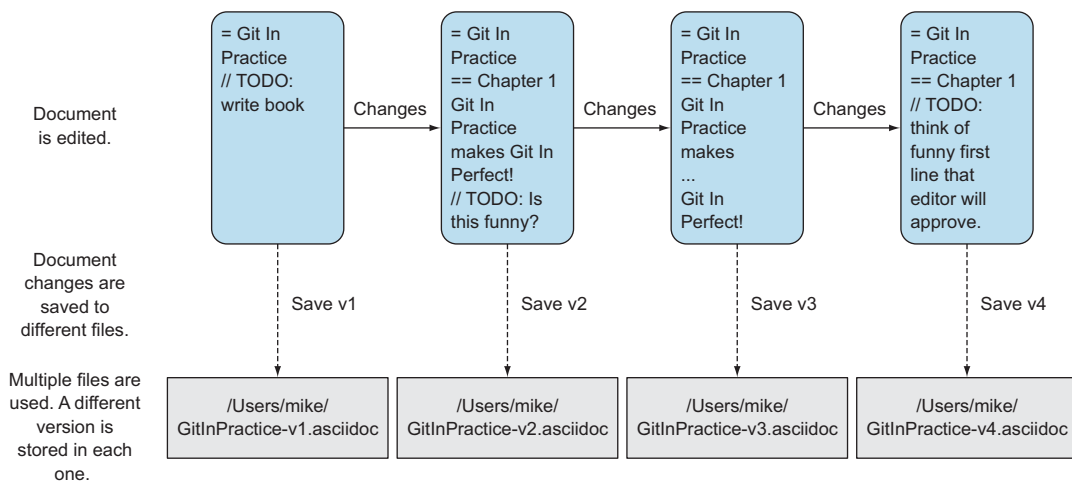


Figure D.1 Versioning with multiple files

At a more advanced level, you may be sharing a document with other people and, rather than just undoing and redoing changes, wish to know who made a change, why they made it, when they made it and what the change was, and perhaps even store multiple versions of the document in parallel. A *version control system* (such as Git) allows all these operations and more.

In a version control system, instead of just saving a document after your changes are made, you *commit* it. This involves a save-like operation commanding the version control system to store this particular version and specifying a message stating the reason for the change or what it accomplishes. When another commit is made, the previous version remains in *history*, where its changes can be examined later. Version control systems can therefore solve the problem of reviewing and retrieving previous changes and allow single files to be used rather than duplicated. This workflow is shown in figure D.2.

When editing a file in a version control system, you always edit/save/commit the same file on disk. Its location won't move, either manually or automatically (unless you wish to rename it, of course). When you wish to access previous versions of the file, you can either view them through the version control system or restore the file on disk to a previous version. This allows you to see exactly what may have changed between versions. When using multiple files, you'd have to manually compare each of the files to see differences and keep track of multiple files on your disk.

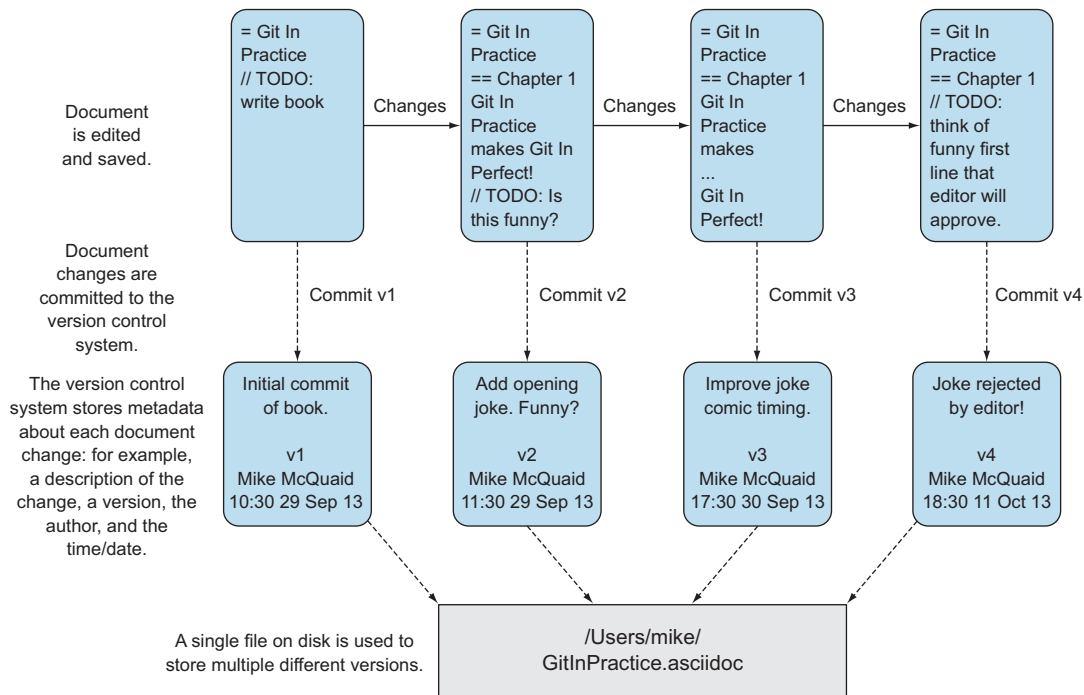


Figure D.2 Versioning with a version control system

D.2 Version control workflow

Version control systems work by maintaining a list of changes to files over time. Each time a file is modified and committed, the new version of the file is stored in the *repository*: a centralized location where the version control system stores files for a particular project. Each commit corresponds to a particular version and stores references to the previously made commit, a *commit message* describing the changes made in this commit, the time it was made, who made it, and the contents of the files at this point. The files' state from a commit can be compared to a previous version, and the difference between the versions' files (known as *diffs*) can be queried.

Figure D.3 shows the workflow used with a version control system. After adding new changes to versioned files, you create new commits containing these changes and commit the changes to the repository. At a later point, you can check out different versions of files. This allows you to have confidence that, no matter what you may add, modify, or delete, all committed versions of your files will remain in the version control system if you need to check their contents later.

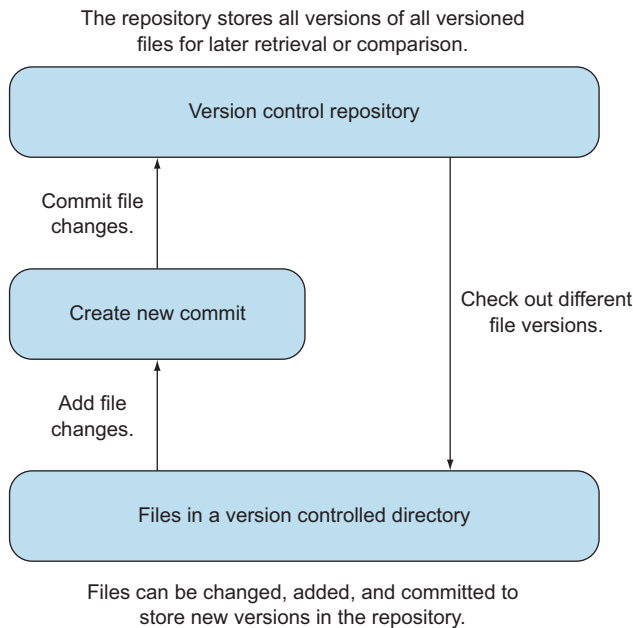


Figure D.3 Git add/commit/checkout workflow

D.3 Version control for programmer collaboration

Programmers spend most of their time editing text. This text is typically source code that will be interpreted by a computer to perform some task, but it could also be software configuration files, documentation, or emails. Because programmers typically work on independent units of work while in larger teams and can be distributed by time or geography, it's important that they communicate explicitly with other pro-

grammers about why a particular change was made. Additionally, programmers inevitably write software that contains bugs. When trying to work out why a bug occurred, it's useful to see what changes were made, by whom, and for what reason. Often programmers need to fix bugs in sections of code they didn't create, so being able to record and recall the intent of the code's author at a later point can help understand what may have caused a bug. These reasons are multiplied by the fact that programmers typically work with huge numbers of source files. Given these reasons, it should be clear why most programming projects use version control systems to manage their source code.

When creating computer software, it's also common to release new versions. New versions are generally released when bugs are fixed and/or when new features have been implemented. Sometimes a team may be developing a new feature but need to provide a new version with a bug fix before the new feature is ready. Two *branches* can be used to work on the bug fix independently of the new feature. This allows work on the bug fix and the new feature to occur in parallel. These branches can later be *merged*, which causes all the changes made in one to be included into the other.

In figure D.4 you can see a simple example of using multiple branches when writing a book.

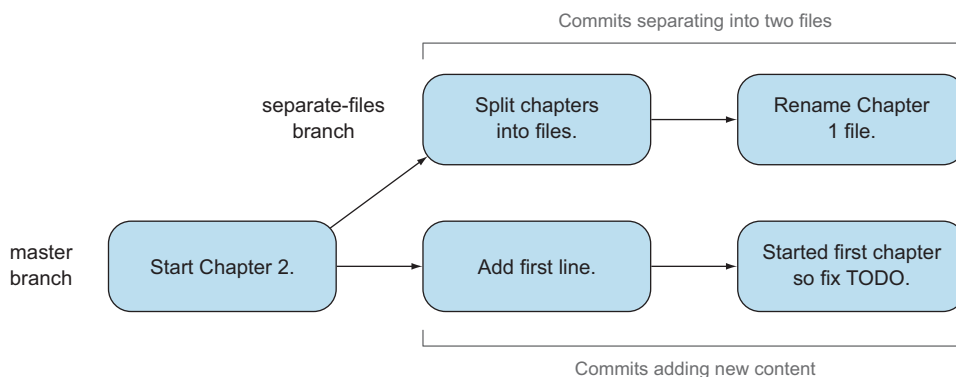


Figure D.4 Committing on multiple branches

Changes, *commits*, *history*, a *repository*, *diffs*, and *branches* are all typically provided by version control systems (such as Git). These features enable workflows where changes are logged for future reference, work can be done in parallel, and previous versions of files are kept. Hopefully this provides you with a basic understanding of why version control systems are useful.

index of Git methods

gh 166–167
gh clone 165
gh fork 168
gh merge 172
gh pull-request 165, 168
git add 7–9, 91
git add --patch 188–189, 191
git bisect 78–82
git bisect bad 82–83
git bisect good 82–83
git bisect log 81
git bisect run 82–83
git bisect skip 81
git blame 76–78
git branch 37, 39–40, 49–50, 86, 91
git checkout 40–42, 110
git cherry 102–103
git cherry-pick 98–100
git clean 57–58, 61
git clone 30–32
git clone --mirror 176–177
git commit 7, 10–12
git commit --patch 190–191
git config 130–137
git daemon 178–180
git describe 97–98
git diff 18–21, 195
git diff --check 194–195
git fetch 35–37
git filter-branch 121–125, 159
git help 135
git init 5–6
git init --bare 175
git instaweb 180–182
git log 13, 15, 17–18, 69–76, 106
git ls-files 65–66
git merge 44–46, 86–87, 90, 93, 170
git mergetool 93
git mv 54–55
git pull 32–34, 36–37, 118
git pull --rebase 117
git push 28–30, 42–43, 48–49, 96, 133
git push --force 119
git rebase 110–113
git rebase --interactive 115–117, 171
git reflog 105–106, 109, 112
git remote 27
git remote add 24–27
git remote prune 134
git rerere 94–95
git reset 56–57, 107–110
git revert 101–102
git rev-parse 23
git rm 55–56
git shortlog 75
git show 70–71, 89, 101, 145
git show master 91–92
git stash 61–62
git stash clear 64
git stash pop 63
git submodule 142
git submodule add 143–146

- git submodule deinit 147
- git submodule foreach 149–150
- git submodule init 147
- git submodule status 146–147
- git submodule update 147–149
- git subtree 142
- git svn 152–155, 158
- git svn clone 152, 158
- git svn dcommit 157, 159–160
- git svn fetch 156–157
- git svn rebase 156–157
- git svn show-ignore 156
- git tag 95–96
- git update-index 65, 67
- git --version 221–223
- git web--browse 135
- gitk 13, 17
- gitx 13, 17

A

- add command
 - adding remote repository 24–27
 - adding submodules 143–146
 - committing only parts of files 188–191
 - overview 8–9
- aliasing commands 137–138
- apt-get command 222
- Arch Linux 222
- authoritative versions 27–28
- authors
 - committer vs. 73
 - Subversion integration 158
- autocorrection feature 136

B

- bare repository 175
- base branch 167
- bisect command
 - automating 82–83
 - overview 78–81
- blame command 76–78
- blob objects 12
- branches
 - adding single commit to current 98–101
 - always creating merge commits 85–87
 - base 167
 - branch command 37–40
 - checking out 40–42
 - collaboration using 237
 - creating local 37–40
 - deleting local after merging 49–50
 - deleting remote 47–49
 - displaying current in terminal 139–140

- feature 197, 207
- head 167
- hotfix 210
- integration 197
- listing branches containing commits 102–103
- merge conflicts
 - avoiding repeated 94–95
 - overview 46–47
 - resolving 88–93
 - using graphical tool 93
- merging
 - overview 44–46
 - rebasing and 47
 - strategies 87–88
- naming 40
- pruning automatically 134
- pull requests and 166
- pulling and rebasing commits 117–119
- purpose of 37
- pushing local 42–43
- rebasing commits 110–113
- rebasing commits interactively 114–117
- refs and 22
- release 210
- resetting to previous commit 107–110
- rewriting entire history of 121–125
- rewriting history on remote 119–121
- Subversion integration 154–155, 160–161
- tags and
 - creating release versions 95–97
 - generating version number from 97–98
- undoing commit 101–102
- browser, displaying repository 180–182
- bugs, finding commit causing 78–83

C

- caret (^) 77
- centralized version control system. *See* CVS
- cgit 182
- Chacon, Scott 207
- changes
 - fetching from remote repository 35–37
 - pulling from remote repository 32
 - pushing to remote repository 28–30
 - showing last in files 76–78
 - stashed
 - clearing 64
 - reapplying 63
 - temporarily stashing 61–63
- checking out
 - checkout command 40–42
 - cloning vs. 31
 - Git vs. Subversion 41
 - reset command vs. 110
 - with uncommitted changes 42
- cherry command 101–103
- cherry-pick command 98–101
- clean command 57–59
- clean history
 - avoiding whitespace issues 194–195
 - building commit in Git Gui 193–194
 - building commit in GitX 191–192
 - commit messages and 187
 - committing only parts of files 188–191
- clear command 64
- cloning repositories
 - checking out vs. 31
 - cloning Subversion repositories 152
 - forking vs. 164
 - mirroring repository 176–178
 - overview 30–32
- CMake
 - commands 198–199
 - pros and cons 202–203
 - workflow using 196–198
- collaboration 236–237
- colored output 133
- commits
 - adding single to current branch 98–101
 - best practices 106–107
 - building in Git Gui 193–194
 - building in GitX 191–192
 - building in staging area 8–9
 - clean history and 187
 - commit command 10–12
 - committer vs. author 73
 - custom output 74
 - defined 7–8
 - displaying single 70–71
 - email format 71–74
 - finding commit that caused bug 78–83
 - graph output 75–76
 - keeping small 14
 - listing branches containing 102–103
 - listing certain 69–70
 - merge commits 47, 85–87
 - messages for 14, 187
 - output files and 60
 - overview 10–12
 - parent 13
 - parts of files 188–191
 - pulling branch and rebasing 117–119
 - rebasing interactively 114–117
 - rebasing on branch 110–113
 - release logs 75
 - resetting branch to previous 107–110
 - resetting files to last 56–57
 - showing who last changed each line of file 76–78
 - signing-off 101
 - squashing 18
 - Subversion integration 158–160
 - undoing 101–102
 - viewing differences between
 - formats for 21–22
 - overview 18–20
 - refs 22–23
 - viewing history 13–18
- configuration
 - aliasing commands 137–138
 - arbitrary configuration values 136
 - autocorrection feature 136
 - colored output 133
 - commented example files 229
 - global 130–131
 - global ignore file 134–135
 - help output as HTML 135
 - pruning branches automatically 134
 - push defaults 133–134
 - repository-specific 131–133
 - sharing between machines 138–139
 - showing current branch in terminal 139–140
 - storing passwords in OS X keychain 135–136
- conflicts, merge
 - avoiding repeated 94–95
 - overview 46–47
 - resolving 88–93
 - using graphical tool 93
- continuous deployment 209, 213
- corruption, disk 107
- cut command 66
- CVS (centralized version control system) 24

D

- daemon command 178–180
- dcommit command 159–160
- Debian 222
- deleting
 - ignored files 60–61
 - local branch after merging 49–50
 - remote branch 47–49
 - untracked files 57–59
- delta 18
- dependencies 142
- describe command 97–98
- develop branch 210
- diff command
 - avoiding whitespace issues 194–195
 - formats for 21–22
 - overview 18–20
 - refs 22–23
- diffstat format 18, 21
- disaster recovery
 - avoiding 106–107
 - pulling branch and rebasing commits 117–119
 - rebasings commits interactively 114–117
 - rebasings commits on another branch 110–113
 - resetting branch to previous commit 107–110
 - rewriting entire history of branch 121–125
 - rewriting history on remote branch 119–121
- disk corruption 107
- distributed version control system 25
- dotfiles repository 138, 229
- Driessen, Vincent 209

E

- email address for Git 5
- email format for commits 71–74
- emerge tool 93, 222

F

- fast-forward merge 45, 85
- feature branches 197, 207, 210
- Fedora 222
- fetch command 35–37
- filesystem
 - deleting ignored files 60–61
 - deleting untracked files 57–59
 - ignoring files 59–60
 - removing files 55–56
 - renaming files 54–55
 - resetting files to last commit 56–57
- stashed changes
 - clearing 64
 - reapplying 63
 - temporarily stashing 61–63
- unchanged files
 - assuming files are 64–65

- listing 65–66
 - stop assuming files are 66–67
- filter-branch command 121–125
- filters, Unix 66
- Fink 222
- foreach command 149–150
- forking repositories
 - cloning vs. 164
 - defined 163–165
 - merging pull requests 172–173
 - using gh tool 168–169
- FreeBSD 222
- fsck tool 107

G

- gc command 109
- Gentoo 222
- gh -b 167
- gh -h 167
- gh tool
 - fork requests using 168–169
 - merging pull requests from forked repository 172–173
 - overview 165
 - pull requests using 165–168
- Git
 - advantages of using 3–4
 - authoritative versions and 27–28
 - checkout terminology 41
 - help 6
 - installing
 - Linux or Unix 222
 - Microsoft Windows 223
 - OS X Mavericks or newer 221
 - OS X Mountain Lion or older 222
 - verifying installation 223
 - object store 12–13
 - setting up 4–5
 - Subversion integration
 - authors and committers 158
 - branches 154–155, 160–161
 - commits 159–160
 - ignore rules 155–156
 - importing repository 151–154
 - migrating repository to Git 158–159
 - pushing to repository 159–160
 - tags 154–155, 160–161
 - updating repository 156–158
 - viewing repository in GitX 158
- git add --force 55
- git add --patch 188–189
- Git Bash 223
- git blame ^ (caret) 77
- git blame --date 77

- git blame -e 78
- git blame -L 78
- git blame -s 78
- git blame --show-email 78
- git blame -w 78
- git branch --delete 49
- git cherry -v 103
- git cherry --verbose 103
- git cherry-pick --abort 100
- git cherry-pick --continue 100
- git cherry-pick --edit 100
- git cherry-pick -s 100
- git cherry-pick -x 100
- git clean --dry-run 58
- git clean --force 58
- git clean --force -X 61
- git clean -xdf 61
- git clone --bare 32, 177
- git clone --depth 32
- git clone --mirror 32, 177–178
- git clone --recurse-submodules 32, 149
- git clone --recursive-submodules 149
- git command. *See individual commands*
- git commit --amend 110, 171
- git commit --author 12
- git commit --date 12
- git commit -i 190
- git commit --message 12
- git commit --patch 190–191
- git config --file 131
- git config --global 130–131
- git config --local 131
- git config --system 131
- git daemon --base-path 179
- git daemon --enable 180
- git daemon --verbose 179
- git describe --tags 98
- git diff --check 194–195
- git diff --stat 21
- git diff --word-diff 21
- git filter branch --all 124
- git filter branch --commit-filter 125
- git filter branch --env-filter 124
- git filter branch --force 124
- git filter branch --index-filter 123
- git filter branch --msg-filter 125
- git filter branch --parent-filter 125
- git filter branch --prune-empty 123
- git filter branch --subdirectory-filter 125
- git filter branch --tag-name-filter 125
- git filter branch --tree-filter 124
- Git Flow
 - extensions for 212
 - overview 209–212
 - pros and cons 212–213
- git help --help 135
- git init --bare 175
- git init --help 6
- git instaweb --local 182
- git instaweb --port 182
- git log --abbrev-commit 106
- git log --after 70
- git log %an 74
- git log %ar 74
- git log --author 69
- git log --before 70
- git log --date 73
- git log --format 72–73
- git log --graph 75–76
- git log --grep 70
- git log --max-count 70
- git log --merges 70
- git log --min-parents 70
- git log -n 70
- git log --patch 18, 73
- git log --pretty 73, 106
- git log --reverse 70
- git log %s 74
- git log --stat 18
- git log --until 70
- git log --walk-reflogs 106
- git log --word-diff 18
- git ls-files --others 58
- git ls-files -v 66
- git merge --ff-only 87
- git merge --no-ff 87
- git merge -s 87
- git merge --strategy 87
- git mv -f 55
- git pull --rebase 34, 117–119
- git push --all 30
- git push --delete 48
- git push --force 30, 121
- git push --set-upstream 28–29
- git push --tags 97
- git push -u 29
- git rebase --abort 113
- git rebase --continue 113
- git rebase -interactive 114–117, 171
- git rebase --skip 113
- git reset --hard 57, 107, 110
- git reset --mixed 107
- git reset --mixed 57, 107, 110
- git reset --soft 110
- git rm --cached 123
- git rm --dry-run 56, 58
- git rm --ignore-unmatch 123
- git rm -n 56
- git rm -r 56
- .git subdirectory 6–7

- git submodule add --force 146
- git submodule add --quiet 146
- git submodule --depth 146
- git submodule foreach --quiet 150
- git submodule foreach --recursive 150
- git submodule status --recursive 147
- git submodule update --depth 149
- git submodule update --force 149
- git submodule update --init 147–149
- git submodule update --no-fetch 149
- git submodule update --recursive 149
- git svn clone --author-file 158
- git svn --mergeinfo 160
- git tag -d 96
- git tag --delete 96
- git tag -f 96
- git tag --force 96
- git tag -l 96
- git tag --list 96
- git update-index --no-assume-unchanged 67
- git update-index --assume-unchanged 65
- git --version 133, 221
- git://protocol 28, 178
- .gitattributes file 229
- .gitconfig file 131, 138, 229
- .gitconfig-user file 229
- git-credential-winstore tool 136
- GitHub
 - creating repository 227–228
 - gh tool
 - fork requests using 168–169
 - merging pull requests from forked repository 172–173
 - overview 165
 - pull requests using 165–168
 - Git and 4
 - help pages on pull requests and forks 164
 - hosting using 182
 - setting up account 225–226
- GitHub Flow
 - overview 207–208
 - pros and cons 209
- .gitignore file 59–60, 135, 155–156, 229
- gitk tool 15, 193
- .gitmodules file 145
- gitolite 182
- git-svn package 152
- gitweb 180
- GitX tool 15–17, 158, 191–192
- global configuration 130–131
- global ignore file 134–135
- grep command 66
- gui command 193–194
- gvimdiff tool 93

H

- head branch 167
- HEAD ref 22
- help output as HTML 135
- history, commit
 - custom output 74
 - email format 71–74
 - finding commit that caused bug 78–83
 - graph output 75–76
 - Homebrew and 203
 - listing all including rewrites 105–106
 - listing only certain commits 69–70
 - maintaining clean
 - avoiding whitespace issues 194–195
 - building commit in Git Gui 193–194
 - building commit in GitX 191–192
 - commit messages and 187
 - committing only parts of files 188–191
 - release logs 75
 - rewriting
 - entire history of branch 121–125
 - pulling branch and rebasing
 - commits 117–119
 - rebasing commits interactively 114–117
 - rebasing commits on another branch 110–113
 - on remote branch 119–121
 - resetting branch to previous commit 107–110
 - showing who last changed each line of file 76–78
 - single commit 70–71
 - squashing commits 18
 - viewing 13–18
- \$HOME directory 130
- Homebrew
 - commands 201–202
 - installing Git 222
 - pros and cons 203–204
 - readable history 203
 - workflow using 200
- hosting repositories
 - displaying repository in browser 180–182
 - initializing in server hosting format 175–176
 - mirroring repository 176–178
 - sharing repository with others on network 178–180
 - tools for 182
- hotfix branches 210
- HTTP (Hypertext Transfer Protocol) 28

I

- ignore property 155
- ignore-all-space option 88

- ignored files
 - adding 59–60
 - deleting 60–61
 - global ignore file 134–135
- init command
 - initializing repository in server hosting
 - format 175–176
 - overview 5–6
- initial commit 13
- installing
 - on Linux or Unix 222
 - on Microsoft Windows 223
 - on OS X Mavericks or newer 221
 - on OS X Mountain Lion or older 222
 - verifying installation 223
- instaweb command 180–182
- integration branches 197
- issue trackers 163

K

- kdiff3 tool 93
- Kitware 196

L

- Linux kernel project and Git 3
- Linux, installing Git on 222
- log command
 - email format 71–74
 - listing only certain commits 69–70
 - overview 13–15
- ls-files command 58

M

- Mac OS X
 - installing on Mavericks or newer 221
 - installing on Mountain Lion or older 222
 - passwords in keychain 135–136
- master branch
 - in Git Flow 209
 - merging pull requests and 164
- master ref 22
- matching push strategy 133
- meld tool 93
- merge commit
 - cherry-pick command vs. 100
 - overview 47
- merge conflicts
 - avoiding repeated 94–95
 - overview 46–47
 - resolving 88–93
 - using graphical tool 93
 - who should resolve 93
- merge strategies 87–88
- mergetool command 93
- merging branches

- deleting local branch afterwards 49–50
- merge command 44–47
- overview 44–46
- rebasing 47
- merging pull requests
 - from forked repository 172–173
 - from same repository 169–171
- merging vs. rebasing
 - CMake
 - commands 198–199
 - pros and cons 202–203
 - workflow using 196–198
 - Homebrew
 - commands 201–202
 - pros and cons 203–204
 - workflow using 200
 - selecting strategy 204–205
- messages, commit 14
- Microsoft Windows
 - installing Git 223
 - instaweb command and 180
 - storing credentials in 136
- Mike Flow
 - Mike Flow Multiple 215–217
 - Mike Flow Single 213–215
 - overview 213
 - pros and cons 217–218
- mirroring repository 176–178
- mv command 54–55

N

- naming
 - bare repositories 176
 - branches 40
- next branch 197
- nightly branch 197

O

- object store 12–13
- octopus merge strategy 87
- OpenBSD 222
- opendiff tool 93
- origin ref 27
- ours merge strategy 87
- ours option 88
- output files 60

P

- packed-refs file 176, 178
- pacman command 222
- parent commits 13
- passwords 135–136
- patience option 88
- personalization
 - aliasing commands 137–138

- personalization (*continued*)
 - arbitrary configuration values 136
 - autocorrection feature 136
 - colored output 133
 - global configuration 130–131
 - global ignore file 134–135
 - help output as HTML 135
 - pruning branches automatically 134
 - push defaults 133–134
 - repository configuration 131–133
 - sharing configuration between machines 138–139
 - showing current branch in terminal 139–140
 - storing passwords in OS X keychain 135–136
- pkg command 222
- placeholders for custom output 74
- pop command 63
- private repositories 226
- pruning branches
 - automating 134
 - prune command 27
- public repositories 226
- pull command
 - fetch command vs. 37
 - overview 32
 - rebasing commits 117–119
- pull requests
 - branches and 166
 - closing without merge 171
 - defined 163–165
 - from forked repository 168–169
 - GitHub and 224
 - merging from forked repository 172–173
 - merging from same repository 169–171
 - using gh tool 165–168
 - viewing without whitespace changes 168
- pushing changes
 - defaults for 133–134
 - overview 28–30
 - pushing local branch 42–43
 - rewriting history on remote branch 119–121
- R**
- rebasing
 - on branch 110–113
 - interactively 114–117
 - merging vs.
 - CMake workflow 196–199, 202–203
 - Homebrew workflow 200–204
 - selecting strategy 204–205
 - overview 47
 - pulling remote branch and 117–119
 - rebase command 110–113
 - Subversion integration 156
- recursive merge strategy 87
- reference log 105
- reflog command 105–106
- refs
 - tags and 95
 - using with diff command 22–23
- regressions 78
- regular expressions 70
- release branches 210
- release versions 95–97
- remote repositories
 - adding 24–27
 - authoritative versions and 27–28
 - checking out local branch 40–42
 - cloning 30–32
 - creating new local branch 37–40
 - deleting local branch after merging 49–50
 - deleting remote branch 47–49
 - fetching changes from 35–37
 - merging branch
 - merge conflicts 46–47
 - overview 44–46
 - rebasing 47
 - pulling changes from 32
 - pushing changes to 28–30
 - pushing local branch 42–43
 - updating tags 97
- remove command 123
- renaming files 54–55
- repositories
 - bare 175
 - committing changes 7–8, 10–12
 - configuration for 131–133
 - creating on GitHub 227–228
 - defined 5, 236
 - hosting
 - displaying repository in browser 180–182
 - initializing in server hosting format 175–176
 - mirroring repository 176–178
 - sharing repository with others on
 - network 178–180
 - tools for 182
 - initializing 5–6
 - private 226
 - public 226
 - remote
 - adding 24–27
 - authoritative versions and 27–28
 - checking out local branch 40–42
 - cloning 30–32
 - creating new local branch 37–40
 - deleting local branch after merging 49–50
 - deleting remote branch 47–49
 - fetching changes from 35–37

- repositories, remote (*continued*)
 - merge conflicts 46–47
 - merging branch 44–46
 - pulling changes from 32
 - pushing changes to 28–30
 - pushing local branch 42–43
 - rebasing 47
 - updating tags 97
- staging area 8–9
- Subversion
 - importing into Git 151–154
 - pushing to 159–160
 - updating 156–158
 - viewing history 13–18
- rerere command 94–95
- reset command 56–57, 107–110
- revert command 101–102, 202
- rev-parse command 23
- rewriting history
 - entire history of branch 121–125
 - pulling branch and rebasing commits 117–119
 - rebasing commits interactively 114–117
 - rebasing commits on another branch 110–113
 - on remote branch 119–121
 - resetting branch to previous commit 107–110
- rm command 55–56
- Ruby 180
- run command 82–83
- S**
- save argument 62
- Secure Shell. *See* SSH
- SHA-1 hash
 - cherry-pick command and 99
 - commit references 10
 - defined 11
- sharing configuration 138–139
- shortlog command 75
- show command 27, 70–71
- signing-off commits 101
- simple push strategy 133
- SMB protocol 178
- Solaris 11 Express 222
- squashing commits 18
- SSH (Secure Shell) 28, 199
- stacks 62
- stage command 199
- start point 39
- stashed changes
 - clearing 64
 - overview 61–63
 - reapplying 63
 - reflog command and 106
 - stash command 61–63
- status command
 - committing only parts of files 189
 - for submodules 146–147
- SubGit 162
- submodules
 - adding 143–146
 - displaying status of 146–147
 - running commands in 149–150
 - updating 147–149
 - when to use 141–142
- subtree command 142
- subtree merge strategy 87
- Subversion (SVN)
 - accessing GitHub repository with 161–162
 - authoritative versions and 27–28
 - branches and 38
 - centralized version control 5
 - checkout terminology 41
 - integration
 - authors and committers 158
 - branches 154–155, 160–161
 - commits 159–160
 - documentation 162
 - ignore rules 155–156
 - importing repository 151–154
 - migrating repository to Git 158–159
 - pushing to repository 159–160
 - tags 154–155, 160–161
 - updating repository 156–158
 - viewing repository in GitX 158
 - speed of 4
- symbolic-ref command 139
- T**
- tags
 - creating release versions 95–97
 - generating version number from 97–98
 - Subversion integration 154–155, 160–161
 - updating remote repository 97
- team workflows
 - Git Flow
 - overview 209–212
 - pros and cons 212–213
 - GitHub Flow
 - overview 207–208
 - pros and cons 209
 - Mike Flow
 - Mike Flow Multiple 215–217
 - Mike Flow Single 213–215
 - overview 213
 - pros and cons 217–218
 - selecting strategy 218–219
 - version control systems advantages 236
- theirs option 88
- Tigerbrew 222
- topic branches 197

tortoisemerge tool 93
Torvalds, Linus 3
tracking branch 30
tree object 12

U

Ubuntu 222
unchanged files
 assuming files are 64–65
 listing 65–66
 stop assuming files are 66–67
unified format diff 21
Unix 66, 222
untracked files 57–59
update command 147–149
update-index command 65

V

vendor dependencies 142
version control system 3
 collaboration 236–237
 handling changes 234–235
 workflow using 236
version number 97–98
vimdiff tool 93

W

WEBrick 180
whitespace

avoiding issues with 194–195
viewing pull request without changes to 168
workflows
 CMake
 commands 198–199
 overview 196–198
 pros and cons 202–203
 Git Flow
 overview 209–212
 pros and cons 212–213
 GitHub Flow
 overview 207–208
 pros and cons 209
 Homebrew
 commands 201–202
 overview 200
 pros and cons 203–204
 Mike Flow
 Mike Flow Multiple 215–217
 Mike Flow Single 213–215
 overview 213
 pros and cons 217–218
 selecting strategy 204–205, 218–219
 using Git 7
 version control system advantages 236

Y

yum command 222

Git IN PRACTICE

Mike McQuaid

Git is a source control system, but it's a lot more than just that. For teams working in today's agile, continuous delivery environments, Git is a strategic advantage. Built with a decentralized structure that's perfect for a distributed team, Git manages branching, committing, complex merges, and task switching with minimal ceremony so you can concentrate on your code.

Git in Practice is a collection of battle-tested techniques designed to optimize the way you and your team manage development projects. After a brief overview of Git's core features, this practical guide moves quickly to high-value topics like history visualization, advanced branching and rewriting, optimized configuration, team workflows, submodules, and how to use GitHub pull requests. Written in an easy-to-follow Problem/Solution/Discussion format with numerous diagrams and examples, it skips the theory and gets right to the nitty-gritty tasks that will transform the way you work.

What's Inside

- Team interaction strategies and techniques
- Replacing bad habits with good practices
- Juggling complex configurations
- Rewriting history and disaster recovery

Written for developers familiar with version control and ready for the good stuff in Git.

Mike McQuaid is a software engineer at GitHub. He's contributed to Qt and the Linux kernel, and he maintains the Git-based Homebrew project.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/GitInPractice



“Shows how to make your team's workflows simpler and more effective.”

From the Foreword by
Scott Chacon, Author of *Pro Git*

“The best companion for your day-to-day journeys with Git.”

—Gregor Zurowski, Sotheby's

“Ready to take off your Git training-wheels? Read this book!”

—Patrick Toohey
Mettler-Toledo Hi-Speed

“I learned more about how Git works in the first five chapters than I did in five years of using Git!”

—Alan Lenton, Arithmetica Ltd



MANNING

\$39.99 / Can \$41.99 [INCLUDING eBook]

ISBN 13: 978-1-61729-197-5
ISBN 10: 1-61729-197-8



9 781617 291975