Join the discussion @ p2p.wrox.com

Wrox Programmer to Programmer™

Professional Git®

Brent Laster

PROFESSIONAL GIT®

INTRODUCTION xxiii

▶ PART I UNDERSTANDING GIT CONCEPTS

CHAPTER 1 What Is Git? 3

CHAPTER 2 Key Concepts 19

CHAPTER 3 The Git Promotion Model 31

CONNECTED LAB 1 Installing Git 43

▶ PART II USING GIT

CHAPTER 4 Configuration and Setup 49

CHAPTER 5 Getting Productive 73

CONNECTED LAB 2 Creating and Exploring a Git Repository and Managing Content 99

CHAPTER 6 Tracking Changes. 105

CONNECTED LAB 3 Tracking Content through the File Status Life Cycle 127

CHAPTER 7 Working with Changes over Time and Using Tags 131

CONNECTED LAB 4 Using Git History, Aliases, and Tags 155

CHAPTER 8 Working with Local Branches 159

CONNECTED LAB 5 Working with Branches 189

CHAPTER 9 Merging Content 193

CONNECTED LAB 6 Practicing with Merging. 243

CHAPTER 10 Supporting Files in Git. 245

CHAPTER 11 Doing More with Git 263

CONNECTED LAB 7 Deleting, Renaming, and Stashing 317

CHAPTER 12 Understanding Remotes—Branches and Operations 321

CONNECTED LAB 8 Setting Up a GitHub Account
and Cloning a Repository. 353

CHAPTER 13 Understanding Remotes—Workflows for Changes 355

CONNECTED LAB 9 Using the Overall Workflow
with a Remote Repository 377

CHAPTER 14 Working with Trees and Modules in Git 381

CONNECTED LAB 10 Working with Worktrees 413

CONNECTED LAB 11 Working with Submodules. 415

CONNECTED LAB 12 Working with Subtrees. 419

CHAPTER 15 Extending Git Functionality with Git Hooks. 423

INDEX 443

PROFESSIONAL

Git®

Brent Laster

wrox™
A Wiley Brand

Professional Git®

Published by
John Wiley & Sons, Inc.
10475 Crosspoint Boulevard
Indianapolis, IN 46256
www.wiley.com

Copyright © 2017 by John Wiley & Sons, Inc., Indianapolis, Indiana

Published simultaneously in Canada

ISBN: 978-1-119-28497-0
ISBN: 978-1-119-28498-7 (ebk)
ISBN: 978-1-119-28500-7 (ebk)

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 646-8600. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008, or online at http://www.wiley.com/go/permissions.

Limit of Liability/Disclaimer of Warranty: The publisher and the author make no representations or warranties with respect to the accuracy or completeness of the contents of this work and specifically disclaim all warranties, including without limitation warranties of fitness for a particular purpose. No warranty may be created or extended by sales or promotional materials. The advice and strategies contained herein may not be suitable for every situation. This work is sold with the understanding that the publisher is not engaged in rendering legal, accounting, or other professional services. If professional assistance is required, the services of a competent professional person should be sought. Neither the publisher nor the author shall be liable for damages arising herefrom. The fact that an organization or Web site is referred to in this work as a citation and/or a potential source of further information does not mean that the author or the publisher endorses the information the organization or Web site may provide or recommendations it may make. Further, readers should be aware that Internet Web sites listed in this work may have changed or disappeared between when this work was written and when it is read.

For general information on our other products and services please contact our Customer Care Department within the United States at (877) 762-2974, outside the United States at (317) 572-3993 or fax (317) 572-4002.

Wiley publishes in a variety of print and electronic formats and by print-on-demand. Some material included with standard print versions of this book may not be included in e-books or in print-on-demand. If this book refers to media such as a CD or DVD that is not included in the version you purchased, you may download this material at http://booksupport.wiley.com. For more information about Wiley products, visit www.wiley.com.

Library of Congress Control Number: 2016956722

Trademarks: Wiley, the Wiley logo, Wrox, the Wrox logo, Programmer to Programmer, and related trade dress are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates, in the United States and other countries, and may not be used without written permission. Git is a registered trademark of Software Freedom Conservancy, Inc. All other trademarks are the property of their respective owners. John Wiley & Sons, Inc., is not associated with any product or vendor mentioned in this book.

To my soul mate and wife, Anne-Marie,

who has taught me what it means to have your dreams come true.

And to my boys Walker, Chase, and Tanner,

who inspire me every day to do the best I can.

ABOUT THE AUTHOR

BRENT LASTER is a senior manager, software development, in the Research and Development Division at SAS in Cary, North Carolina. He manages several groups involved with release engineering processes and internal tooling. He also serves as a resource for the use of open-source technologies and conducts internal training classes in technologies such as Git, Gerrit, Gradle, and Jenkins, both in the U.S. and abroad.

In addition to corporate training, Brent creates and presents workshops for a wide variety of technical conferences. He has presented workshops and informational sessions on open-source technologies (and how to apply them) at such conferences as the Rich Web Experience/Continuous Delivery Experience, ÜberConf, OSCON, and others. He is also a contributor to publications such as *No Fluff Just Stuff* magazine. Brent also conducts live web training from time to time.

Brent's passion is teaching, and he does so in a way that makes difficult concepts relatable to all. He has been involved in technical training for over 25 years and continues to seek out ways to show others how technology can be used to simplify and automate their work.

You can learn more about Brent and his work on his LinkedIn page at `http://linkedin.com/in/BrentLaster` or on Twitter at `@BrentCLaster`.

ABOUT THE TECHNICAL EDITOR

CHAIM KRAUSE is a Simulation Specialist for the U.S. Army in Leavenworth, Kansas. Although he holds a BA in Political Science from the University of Chicago, Chaim is an autodidact when it comes to computers, programming, and electronics. He wrote his first computer game in BASIC on a TRS-80 Model I Level I and stored the program on a cassette tape. Amateur radio introduced him to electronics, while the Arduino and the Raspberry Pi provided a medium to combine computing, programming, and electronics into one hobby.

In his spare time, he likes to play PC games and occasionally develops his own. He has recently taken up the sport of golf to spend more time with his significant other, Ivana.

ABOUT THE TECHNICAL PROOFREADER

PHILIPPE CHARRIÈRE is a solution engineer for GitHub. He is also a speaker at many technical events about programming, including JDD (Poland), Devoxx France, SoftShake (Geneva), and Voxxed Days Luxembourg. He started programming with a TI-99/4A (Texas Instruments) a long time ago. In his spare time, he develops IOT solutions.

CREDITS

PROJECT EDITOR
Adaobi Obi Tulton

TECHNICAL EDITOR
Chaim Krause

TECHNICAL PROOFREADER
Philippe Charrière

PRODUCTION EDITOR
Barath Kumar Rajasekaran

COPY EDITOR
Marylouise Wiack

MANAGER OF CONTENT DEVELOPMENT & ASSEMBLY
Mary Beth Wakefield

PRODUCTION MANAGER
Kathleen Wisor

MARKETING MANAGER
Carrie Sherrill

PROFESSIONAL TECHNOLOGY & STRATEGY DIRECTOR
Barry Pruett

BUSINESS MANAGER
Amy Knies

EXECUTIVE EDITOR
Jim Minatel

PROJECT COORDINATOR, COVER
Brent Savage

PROOFREADER
Nancy Bell

INDEXER
Johnna VanHoose

COVER DESIGNER
Wiley

COVER IMAGE
Doug Lemke/Shutterstock

ACKNOWLEDGMENTS

Over the course of this project, I have been amazed to discover how many dedicated people it takes to produce a book of this sort. The team at John Wiley & Sons has been outstanding and deserves more thanks than I can offer here.

First, thanks to Jim Minatel for taking on this project and seeing the potential for a new book on Git. I've appreciated his vision for the project as well as his practical guidance. Thanks also to Adaobi Obi Tulton, an amazing project editor who provided excellent direction and advice while juggling what seemed to be a million details, and never dropped the ball. This book is only possible because of her dedicated management and assistance. Thanks to Marylouise Wiack the copy editor, for making my writing readable and clear; Barath Kumar Rajasekaran, the production editor; and Nancy Bell, the proofreader, for bringing everything together to create a final, polished product.

Sincere thanks to Chaim Krause, the technical editor, for all of the effort and time he put in to making sure my explanations, examples, and labs were correct and made sense. I appreciated his careful attention to detail, and his suggestions based on his own experience toward making this book even more useful to readers. Thanks also to Philippe Charrière for volunteering on short notice to be a second technical reviewer. I've appreciated the commitment, comments, and suggestions that he has provided as an experienced professional.

Thanks to the management at SAS for supporting my initiatives to create and present corporate training courses over the years to employees across the company. I especially thank Glenn Musial, Cyndi Schnupper, and Andy Diggelmann for their encouragement and positive feedback; Barbara Miller for the years she spent managing all the logistics of scheduling and preparing materials for the training sessions as they expanded into so many areas and internationally; and my colleague, Lee Greene, for helping to teach the classes and review the materials.

On the conference side, a big thanks to Jay Zimmerman, the founder and organizer of the No Fluff Just Stuff conference series, for giving me the opportunity to speak at their events all across the country. It's great to be able to participate in quality technical events like these, which provide meaningful information and training for all.

Thanks also to everyone who's attended one of my training sessions or workshops and asked a question or provided feedback. This input can be invaluable in making the content better and more applicable.

An obscure thank you here to my fourth-grade assistant teacher, Ms. King, who told me all those years ago that I should be a writer. I'm not sure this is the kind of book she or I had in mind, but I'm counting it.

Finally, the biggest thanks of all must go to my wife, Anne-Marie, and to my children. This book was written over a long period of time, mostly nights and weekends, which took time away from them. Nonetheless, they never failed in their words of encouragement. Anne-Marie, you have been my inspiration, my friend, and my greatest supporter during this project, even though I know how foreign much of it seemed to you. Thank you for making each day a joy and our life together amazing, and for sharing my dreams and, most of all, my life.

CONTENTS

INTRODUCTION *xxiii*

PART I: UNDERSTANDING GIT CONCEPTS

CHAPTER 1: WHAT IS GIT? 3

History of Git 4
Industry-Standard Tooling 4
The Git Ecosystem 5
Core Git 5
Git-Hosting Sites 6
Self-Hosting Packages 7
Ease-of-Use Packages 8
Plug-ins 10
Tools That Incorporate Git 10
Git Libraries 10
Git's Advantages and Challenges 11
The Advantages 11
The Challenges 14
Summary 18

CHAPTER 2: KEY CONCEPTS 19

Design Concepts: User-Facing 19
Centralized Model 20
Distributed Model 20
Design Concepts: Internal 22
Delta Storage 22
Snapshot Storage 23
Git's Storage Requirements 23
Repository Design Considerations 25
Repository Scope 25
File Scope 27
Summary 29

CHAPTER 3: THE GIT PROMOTION MODEL 31

The Levels of Git 31
Dev-Test-Prod and Git 32

The Core Git Commands for Moving Content 39
Summary 41
About Connected Lab 1: Installing Git 42

CONNECTED LAB 1 INSTALLING GIT 43

Installing Git for Windows 43
Steps 43
Installing Git on Mac OS X 46
Installing Git on Linux 46

PART II: USING GIT

CHAPTER 4: CONFIGURATION AND SETUP 49

Executing Commands in Git 49
Operand Types 50
Porcelain versus Plumbing Commands 51
Specifying Arguments 53
Auto-complete 54
Configuring Git 55
Telling Git Who You Are 56
Configuration Scope 57
Default Editor 60
End of Line Settings 61
Aliases 63
Windows Filesystem Cache 64
Initializing a Repository 64
Git Init 64
Git Clone 65
Advanced Topics 66
Git Init Demystified 67
Looking Further into a Git Repository 67
Mapping Config Commands to Configuration Files 69
Creating Parameterized Aliases 70
Summary 71

CHAPTER 5: GETTING PRODUCTIVE 73

Getting Help 73
The Multiple Repositories Model 76
Adding Content to Track—Add 77
Staging Scope 78

Finalizing Changes—Commit 86
Prerequisites 87
Commit Scope 88
Putting It All Together 88
Amending Commits 88
Results of a Commit 91
Advanced Topics 94
Using the Verbose Option 95
The Full Commit Message Experience 95
Autocorrect and Auto Execute 96
Summary 97
About Connected Lab 2: Creating and Exploring a Git Repository and Managing Content 97

CONNECTED LAB 2 CREATING AND EXPLORING A GIT REPOSITORY AND MANAGING CONTENT 99

Prerequisites 99
Optional Advanced Deep-Dive into the Repository Structure 99
Steps 99

CHAPTER 6: TRACKING CHANGES 105

Git Status 105
Workflow Example with Status 106
Status Command Short Form 110
Git Diff 111
Important Symbolic Names in Git 112
How to Think about Git's Approach to Diffing 112
Diff Names Only 115
Word-diff 116
Ignoring Non-critical Changes 116
Diffing Two Commits 118
Visual Diffing 121
Other Diff Tricks 124
Summary 125
About Connected Lab 3: Tracking Content through the File Status Life Cycle 125

CONNECTED LAB 3 TRACKING CONTENT THROUGH THE FILE STATUS LIFE CYCLE 127

Prerequisites 127
Steps 127

CHAPTER 7: WORKING WITH CHANGES OVER TIME AND USING TAGS 131

The Log Command 132
Common Display and Filtering Options 132
Time-Limiting Options 133
History by Files and Paths 134
Log Output Format 135
Searching History 136
Git Blame 138
Seeing History Visually 140
Tags 141
Seeing Tag Details 142
Modifying Tags 143
Quick Tagging Example 144
Undoing Changes in History 145
Reset—Rolling Back Changes 145
Completely Resetting the Local Environment 146
Revert—Canceling Out Changes 147
Advanced Topics 151
Signing Commits and Tags 151
Reflogs 153
Summary 154
About Connected Lab 4: Using Git History, Tags, and Aliases 154

CONNECTED LAB 4 USING GIT HISTORY, ALIASES, AND TAGS 155

Prerequisites 155
Steps 155

CHAPTER 8: WORKING WITH LOCAL BRANCHES 159

What Is a Branch? 159
Example from Another Source Management System 160
The Git Model of Branches 160
Creating a Branch 162
Checking Out a Branch 163
Adding Content to a Branch 164
One Working Directory—Many Branches 164
Getting Information about Branches 168
Deleting or Renaming a Branch 169
Developing with Branches 172

Advanced Topics 180
Checking Out Non-branch Commits 181
Checking Out Individual Files 186
Summary 187
About Connected Lab 5: Working with Branches 188

CONNECTED LAB 5 WORKING WITH BRANCHES 189

Prerequisites 189
Steps 189

CHAPTER 9: MERGING CONTENT 193

The Basics of Merging 194
The Merge Command 194
Preparing for a Merge 194
Types of Merges 195
Rebasing—Merging with History 199
Cherry-Picking 203
Differences between Cherry-Pick and Rebase 205
Merge Operations 206
Dealing with Conflicts 210
Merging Is a State 210
Error Messages for Conflicts 211
Aborting the Operation 212
Dealing with Conflicts—the Workflow 213
Resolution Options and Strategies 217
Visual Merging 223
Selecting a Merging Tool 226
Making Merge Tools Available to Git 226
Advanced Topics 227
Alternative Style for Conflict Markers 227
Advanced Rebasing Scenario 229
Interactive Rebasing 234
Summary 241
About Connected Lab 6: Practicing with Merging 242

CONNECTED LAB 6 PRACTICING WITH MERGING 243

Prerequisites 243
Steps 243

CHAPTER 10: SUPPORTING FILES IN GIT 245

The Git Attributes File 246
What Can You Do with This File? 246
The Scope of Git Attributes 246
The File Format 248
Common Use Cases 249
Getting Attribute Information for Files 256
The Git Ignore File 256
The Scope of Git Ignore 257
The File Format 258
Getting Ignore Information for Files 260
Summary 261

CHAPTER 11: DOING MORE WITH GIT 263

Modifying the Layout of Files and Directories in Your Local Environment 263
stash 264
Staging Commands 270
mv 271
rm 272
Commands for Searching 272
grep 273
Git Log Searches 275
Working with Patches and Archives for Changes 278
archive 278
bundle 279
Sharing Patches through E-mail 280
apply 281
am 282
Mailing Patch Files 283
Commands for Cleaning Up 285
clean 286
gc 288
notes 288
Advanced Topics 290
filter-branch 291
rev-list 291
bisect 298
rerere 306
Summary 314
About Connected Lab 7: Deleting, Renaming, and Stashing 315

CONNECTED LAB 7 DELETING, RENAMING, AND STASHING 317

Prerequisites 317
Steps 317

CHAPTER 12: UNDERSTANDING REMOTES—BRANCHES AND OPERATIONS 321

Remotes 322
Remote Access Protocols 324
The Remote Command 326
How Git Interacts with the Remote Environment 328
Remote Tracking Branches 328
Git Clone 330
Viewing Information about Remote Branches 333
Push 338
Fetch 346
Pull 349
Summary 352
About Connected Lab 8: Setting Up a GitHub Account and Cloning a Repository 352

CONNECTED LAB 8 SETTING UP A GITHUB ACCOUNT AND CLONING A REPOSITORY 353

Prerequisites 353
Steps 353

CHAPTER 13: UNDERSTANDING REMOTES—WORKFLOWS FOR CHANGES 355

The Basic Conflict and Merge Resolution Workflow in Git 356
How the Remote Side Handles Conflicts 356
Hosted Repositories 368
Models for Collaboration with Git 369
Putting It All Together 374
Summary 375
About Connected Lab 9: Using the Overall Workflow with a Remote Repository 375

CONNECTED LAB 9 USING THE OVERALL WORKFLOW WITH A REMOTE REPOSITORY 377

Prerequisites 377
Steps 377

CHAPTER 14: WORKING WITH TREES AND MODULES IN GIT 381

Worktrees 381
Adding a Worktree 382
Listing Out the Working Trees 385
Pruning a Worktree 385
Submodules 386
Understanding How Submodules Work 388
Adding a Submodule 388
Determining Submodule Status 389
Processing Multiple Submodules 393
Updating Submodules from Their Remotes 394
Viewing Submodule Differences 396
Superproject versus Submodules 397
The Problem with Submodules 399
Updating the Submodule References 399
Updating Submodules When the Superproject Is Updated 401
Pushing Changes from Submodules 402
Submodules and Merging 404
Unregistering a Submodule 404
Subtrees 405
Adding a Project as a Subtree 407
Updating a Subtree 409
Using the Subtree Split Functionality 410
Creating a New Project from the Split Content 410
Subtree Push 411
Summary 411
About Connected Labs 10–12 411
About Connected Lab 10: Working with Worktrees 412
About Connected Lab 11: Working with Submodules 412
About Connected Lab 12: Working with Subtrees 412

CONNECTED LAB 10 WORKING WITH WORKTREES 413

Prerequisites 413
Steps 413

CONNECTED LAB 11 WORKING WITH SUBMODULES 415

Prerequisites 415
Steps 415

CONNECTED LAB 12 WORKING WITH SUBTREES 419

Prerequisites 419
Steps 419

CHAPTER 15: EXTENDING GIT FUNCTIONALITY WITH GIT HOOKS 423

Installing Hooks 424
Updating Hooks 425
Common Hook Attributes 425
Hook Domain 425
Return Code to Control Workflow 426
Working Directory Access 426
Environment Variables 427
Hook Descriptions 427
Applypatch-msg 427
Pre-applypatch 427
Post-applypatch 428
Pre-commit 428
Prepare-commit-msg 429
Commit-message 431
Post-commit 432
Pre-rebase 433
Post-checkout 434
Post-merge 434
Pre-push 434
Pre-receive 435
Update 436
Post-receive 436
Post-update 437
Other Hooks 438
Push-to-checkout 438
Pre-auto-gc 438
Post-rewrite 438
Hooks Quick Reference 439
Summary 441

INDEX *443*

INTRODUCTION

Welcome. If your job or interests involve designing, creating, or testing software, or managing any part of a software development lifecycle, chances are that you've heard of Git and, at some level, have tried to use and understand it. This book will help you reach that goal. To put it simply, *Professional Git* is intended to help you understand and use Git to get your job done, whether that job is a personal project or a professional requirement. In the process, it will also make Git part of your professional comfort zone. Throughout the book, I've provided the background and concepts that you need to know (and understand) to make sense of Git, while you learn how to interact with it.

This section will provide you with a quick introduction to the book. It will explain how this book is unique from other books about Git, the intended target audience, the book's overall structure and content, and some of the value it offers you.

I encourage you to take a few minutes and read through this section. Then, you can dive into the material at your own pace, and build your skills and understanding of Git through the text and the included hands-on labs. Or, if you'd like to quickly see additional information about the range of content, you can browse the table of contents.

Thanks for taking a look at *Professional Git*.

HOW THIS BOOK IS UNIQUE

While many books about Git are already on the market, most are aimed at providing the technical usage of the application as their major and singular goal. *Professional Git* will provide you with that, but it will also provide you with an understanding of Git in terms of concepts that you probably already know. As well, most books do not provide practical ways to integrate the concepts they describe. Learning is most effective when you have actual examples to work through so you can internalize the concepts and gain proficiency at your pace. *Professional Git* includes Connected Labs that you can work through to absorb what you've just read.

I've included simple, clear illustrations to help you visualize key ideas and workflows. I've also included Advanced Topics sections at the end of many chapters. These sections provide additional explanations of how to use some lesser-known features of Git as well as how to go beyond the standard Git features to gain extra value.

It is easy to experience a bad transition from another source management system to Git, if you don't understand Git. To be most effective, you need to comprehend the Git model and workflow. You should also know what to watch out for as you make the transition and why it's important to consider not only the commands and workflow, but also the structure and scope of its underlying repositories. I cover all of this in *Professional Git*.

TARGET AUDIENCE

This book is based on my years of training people on Git; these people worked at all levels and came from many different backgrounds—developers, testers, project managers, people managers, documentation specialists, and so on. I have presented the basic materials outlined in this book through many workshops at industry conferences and corporate training sessions. I've presented them at locations across the United States, as well as internationally. I've been successful in helping people to walk away with a newfound confidence in using Git.

I only make one assumption in this book: that you have experience with at least one source management system. It doesn't matter which one: CVS, Subversion, Mercury—any will do. I just assume that you have a basic awareness of what a source management system does as well as fundamental concepts such as checking in and checking out code and branching. Beyond that, you do not require any prior knowledge or experience. And even if you have significant experience with Git or another system, you'll find something of benefit here. In fact, if you're reading this, then you probably fall into one of the following categories:

- You are new to Git and know that you need to learn it.
- You have used Git but have been trying to use it the same way you used your previous source control system.
- You have used Git and feel that you know "just enough to be dangerous."
- You are getting by with Git, but really want to understand why it works the way it does and how to really use it as intended.
- You work with, or manage, people who either use Git or need to learn it. Given that association, you need to know about Git and to understand the fundamental concepts.
- You've heard about the potential benefits of Git, and so you are curious about it and about what it can do for you and the organization you work with.

You may actually see yourself in more than one of these categories. However, you probably just want to be able to get your job done (whether that job is a personal or professional goal). This book was built on that premise.

Git requires a mind shift. In fact, it requires a series of mind shifts. However, each shift is easy to understand once you can relate it to something you already know. Understanding each of these shifts will, in turn, allow you to be more productive and to harness the features of this powerful tool—and that's what this book is about.

STRUCTURE AND CONTENT

This book is organized as a series of chapters that present Git from the ground up, teaching you what you need to know and build on to become proficient before adding new concepts.

In the first three chapters, I cover the foundational concepts of Git: how it's different from other systems, the ecosystem that's been built around it, its advantages and challenges, and the model that allows you to understand its workflow and manage content effectively with it. This section will provide you with a basic understanding of the ideas, goals, and essential terminology of Git.

In the remaining chapters of the book, I cover the usage and features of Git, from performing basic operations to create repositories and commit changes into them, to creating branches, doing merges, and working with content in public repositories.

Notice that I don't have you using Git right away. (If you want to do that, feel free to jump ahead to Chapter 4, which quickly enables you to start getting hands-on with Git.) However, I highly recommend reading the first three chapters. If you're new to Git (or it's been a while), the background reading, especially in Chapters 2 and 3, will provide the foundation you need to understand the remaining chapters. And even if you've used Git before, reading these chapters may clear up questions that you've had about Git, give you a better mental model to work from, and form a basis to understand some of the more advanced concepts.

READER VALUE

Throughout the book, you'll find examples and guidance on the commands and workflows you need to be productive with Git. Each chapter includes ways to relate concepts to what you already know and understand. In addition to the text, you'll find many illustrations to help you understand concepts visually. As I've already mentioned, this book also adds a feature that allows you to get hands-on experience with Git, via Connected Labs interspersed throughout the chapters. These labs are designed to reinforce the concepts presented in the text of the preceding chapter(s) and to get you actively involved in the learning process, allowing you to better grasp the concepts. To get the most out of the book, you should take the time to complete each lab—usually only a few minutes. You'll find that these simple steps will greatly increase your overall understanding and confidence when using Git.

As well, take a look at the Advanced Topics sections, located at the end of some chapters. You'll likely find explanations and ideas to leverage Git functionality in ways you may not have considered before, or you may find out how to use that feature you've always wondered about.

For the later labs, custom Git repositories with example content are provided for the user at http://github.com/professional-git. In addition, downloadable copies of the code for the hooks from the last chapter are available in http://github.com/professional-git/hooks. In the event that GitHub is not available, you can find the needed files at www.wrox.com/go/professionalgit

NEXT STEPS

If this sounds like the book for you, then I encourage you to keep reading and to start making the connections and mind shifts that will help you succeed with Git. As you progress through the book, you'll find many ideas, insights, and "a-ha" moments that will serve you well. And with that knowledge, you'll soon be working at the level of "Professional Git."

PART I
Understanding Git Concepts

- **CHAPTER 1:** What Is Git?
- **CHAPTER 2:** Key Concepts
- **CHAPTER 3:** The Git Promotion Model

1

What Is Git?

WHAT'S IN THIS CHAPTER?

- A brief introduction to Git and its history
- The different ways to find and access Git
- Types of applications that incorporate Git
- The advantages of using Git
- The challenges of using Git

In this chapter, you'll be introduced to Git and will learn about it from a product perspective—what it is, why it's used, the different kinds of interfaces you can use with it, and the good parts and challenging parts of working with it. This will provide an important foundation for understanding the technical details that follow in the subsequent chapters.

If I were to summarize what Git is in one paragraph, it would go something like this:

Git is a popular and widely used source management system that greatly simplifies the development cycle. It enables users to create, use, and switch between branches for content development as easily as people create and switch between files in their daily workflow. It is implemented using a fast, efficient architecture that allows for ease of experimentation and refinement of local changes in an isolated environment before sharing them with others. In short, it allows everyday users to focus on getting the content right instead of worrying about source management, while providing more advanced users with the ability to record, edit, and share changes at any level of detail.

In short, Git is different—really. When you're experienced with using Git and understand it, this will make you feel empowered and productive. When you're new to Git, and trying to understand it, you will encounter a model that will lead you to think differently about managing content in source control.

To illustrate, there's an old saying that "when all you have is a hammer, everything looks like a nail." When all you have is a traditional centralized source management system, everything looks like a file-by-file change that is expensive to branch.

Not so with Git. Git is one of those nice tools that actually allows users to focus on developing content and simplifying workflows. It's not just another tool in the toolbox, it is the toolbox. It contains all of the tools you need to manage tracking anything from a few files for a single user to projects spanning hundreds of users and a huge scope, such as the Linux kernel. Today, many large companies use Git. It's free, it's powerful, it scales, and its model works when used as designed.

Git also has a certain "feel" that's appealing to many people. Git is structured more like a series of individual utilities that you can run against your content, similar to how users work with operating systems. However, it doesn't try to be the system; it gives users ultimate control over their content, even to the point of being able to update history if needed. Git manages basic units that equate to directory structures rather than individual files, so content that extends across file and directory boundaries can be managed together. Git simplifies branching, to a point where creating, merging, or deleting branches becomes nearly as quick and easy as creating, merging, or deleting files. It also provides a local environment with full source management control that can be updated independently of the shared, public environment.

Given that it is different from other source code management (SCM) systems, it's useful to understand how Git originated. The following section includes some of its history.

HISTORY OF GIT

Git has its roots in the development environment for the Linux kernel. In the early 2000s, the team working on the kernel began using a proprietary distributed source control system called BitKeeper (sometimes abbreviated as BK). The team was initially allowed to use this system for free. Over time, differences of opinion developed around the use of BK to the point that the owner of that system revoked the free use of the product. At that time (in 2005), Linus Torvalds, the creator of Linux, set out to create a new system that maintained the distributed ideal, but also incorporated several additional concepts he had been working with. Perhaps most importantly, he wanted it to provide the fast performance that a project on the scope of the Linux kernel would need. Thus the motivation and ideas for what became Git came into being.

Development began in early April of 2005, and an initial release was ready by July. Originally, there was an idea of purposing Git as a toolkit that could have other systems implemented on top of it. However, over time, it has been made into a full-fledged SCM in its own right.

If you're wondering about the name, there are multiple definitions for the word *Git,* but all of them imply a negative connotation about a person. Git was given its name by its creator. Linus jokingly stated that he named all his projects after himself.

For those interested in learning more about this phase of Git development, detailed historical information is available on the Internet.

INDUSTRY-STANDARD TOOLING

From these early beginnings, Git has grown to become an industry-standard tool. Of course, *industry standard* is a relative term. Nevertheless, based on nearly any criteria, Git fits. It is used across all levels of industry. Huge projects, such as the Linux kernel, are managed in it, and also mandate

its use (see the following list). It is a key component of many continuous integration/continuous delivery pipelines. Demand for knowledge about it is ever increasing. Commercial and open-source projects and applications recognize that if they require source management services, they have to integrate with Git. Projects and companies using Git include

- Google
- Facebook
- Microsoft
- Twitter
- LinkedIn
- Netflix
- O'Reilly
- PostgreSQL
- Android
- Linux
- Eclipse

As with any sufficiently successful open-source technology, an entire ecosystem has sprung up around Git. This point is worth discussing for a moment. The basic tool that is Git has given rise to a seemingly endless number of applications to further help users who want to work with it—most named with some wordplay based on *git*. If you start discussing Git with someone, you may hear such names as GitHub, Gitolite, Easy Git, Git Extensions, EGit, and so on. To the uninitiated, it can be challenging to understand how each one of these names relates to the original Git tooling. To help clarify some of the confusion, I'll give you an overview of how the different offerings are categorized.

THE GIT ECOSYSTEM

Broadly, you can break down the Git-based offerings into a few categories: core Git, Git-hosting sites, self-hosting packages, ease-of-use packages, plug-ins, tools that incorporate Git, and Git libraries.

Core Git

In the core Git category, you have the basic Git executables, configuration files, and repository management tooling that you can install and use through the command line interface. (These can be installed from https://git-scm.com/downloads.) In addition to the basic pieces, the distributions usually include some supporting tools such as a simple GUI (git gui), a history visualization tool (gitk), and in some cases, an alternate interface such as a Bash shell that runs on Windows. The distribution for Windows is now called Git for Windows. Similarly there is a ported version of Git for OS/X. This version can be installed directly from the git-scm.com site, or via the Homebrew package manager or built via the MacPorts application.

When installing on Linux systems, the recommended method is to use the preferred package manager for your distribution. Example commands are shown in the following list.

- Debian/Ubuntu
  - $ apt-get install git
- Fedora (up to 21)
  - $ yum install git
- Fedora (22 and beyond)
  - $ dnf install git
- FreeBSD
  - $ cd/usr/ports/devel/git
  - $ make install
- Gentoo
  - $ emerge --ask --verbose dev-vcs/git
- OpenBSD
  - $ pkg_add git
- Solaris 11 Express
  - $ pkg install developer/versioning/git

Git-Hosting Sites

Git-hosting sites are websites that provide hosting services for Git repositories, both for personal and shared projects. Customers may be individuals, open-source collaborators, or businesses. Many open-source projects have their Git repositories hosted on these sites. In addition to the basic hosting services, these sites offer added value in the form of custom browsing features, easy web interfaces to Git commands, integrated bug tracking, and the ability to easily set up and share access among teams or groups of individuals.

These sites typically provide a workflow intended to allow users to contribute back to projects on the site. At a high level, this usually involves getting a copy of another user's repository, making changes in the copy, and then requesting that the original user review and incorporate the changes; this is sometimes known as the *fork and pull* model. (This model is explained in more detail in Chapter 13.)

For hosting, there is a pricing model that depends on the level of access, number of users, number of repositories, or features needed. For example, if a repository is intended to be public—with open access to anyone—it may be hosted for free. If access to a repository needs to be limited or it needs a higher level of service, then there may also be a charge. In addition, the hosting site may offer services such as consulting or training to generate revenue.

Examples of these types of sites include GitHub and Bitbucket. Figure 1-1 shows an example of a GitHub repository page.

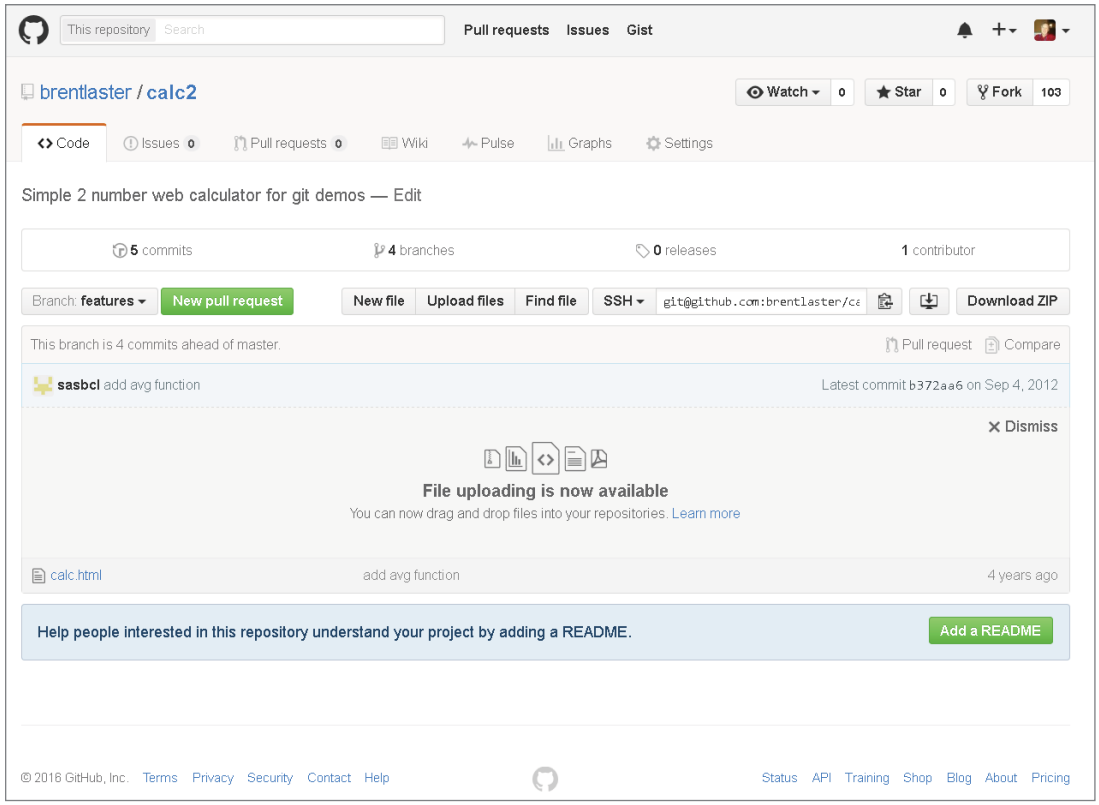


FIGURE 1-1: Example GitHub page

Self-Hosting Packages

Based on the success of the model and usage of the hosting sites, several packages have been developed to provide a similar functionality and experience for users and groups without having to rely on an external service. For some, this is their primary target market (GitLab), while others are stand-alone (also known as *on-premise*) versions of the popular web-hosting sites (such as GitHub Enterprise).

These packages are more palatable to businesses that don't want to host their code externally (on someone else's servers), but still want the collaborative features and control that are provided with the model. The cost structure usually depends on factors relating to the scale of use, such as the number of users or repositories. Figure 1-2 shows an example of a GitLab project screen.

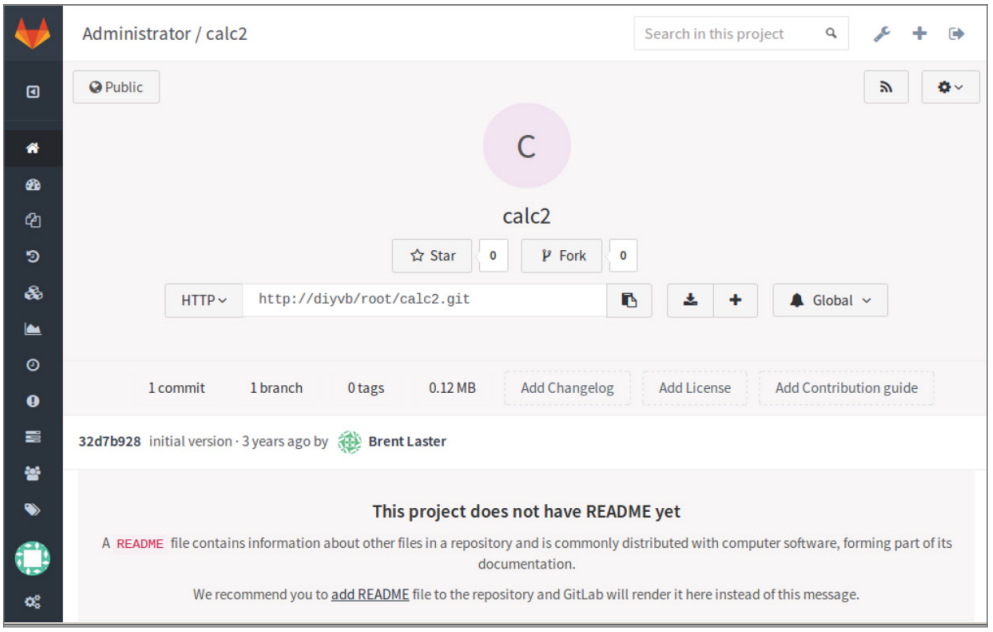


FIGURE 1-2: GitLab project screen

Ease-of-Use Packages

The ease-of-use category encompasses applications that sit on top of the basic Git tooling with the intention of simplifying user interaction with Git. Typically, this means they provide GUI interfaces for working with repositories and may support GUI-based conventions such as drag-and-drop to move content between levels. In the same way, they often provide graphical tools for labor-intensive operations such as merging.

Examples include SourceTree, SmartGit, TortoiseGit, and Git Extensions. Typically, these packages are free for non-commercial use. You can see a more comprehensive list at `https://git-scm.com/downloads/guis`.

Figure 1-3 shows some examples of available packages.

CHOOSING AN INTERFACE

One of the questions that frequently comes up when using Git is which stand-alone interface is best. There is no right answer here, but as a good default, the command line provides the most value for a number of reasons.

Although a large number and variety of GUIs are available to use with Git, there is no accepted standard. GUIs come and go, and vary highly in their degree of functionality, completeness, and utility. The command line is consistent and universally applicable.

Not all functionality is exposed through any one GUI for Git. However, all functionality available to users is exposed through the command line. If you need to do something that isn't available through a GUI, you can always drop back to the command line to accomplish it. In addition, Git includes man pages for all command line usage, so help is readily available for that interface.

If you understand the command line operations and options, it's generally easy to translate and map them to the corresponding items in a GUI.

Once you understand the basic command line operation, you'll have more insight into what you want and need to do with a GUI interface. You'll also be in a better position to choose one if desired.

As a side note, one of the main advantages of having a graphical interface with Git is having a graphical merge tool. Git also allows you to configure using a third-party tool for merges from the command line interface. We'll explore configuring merge tools in Chapter 9.

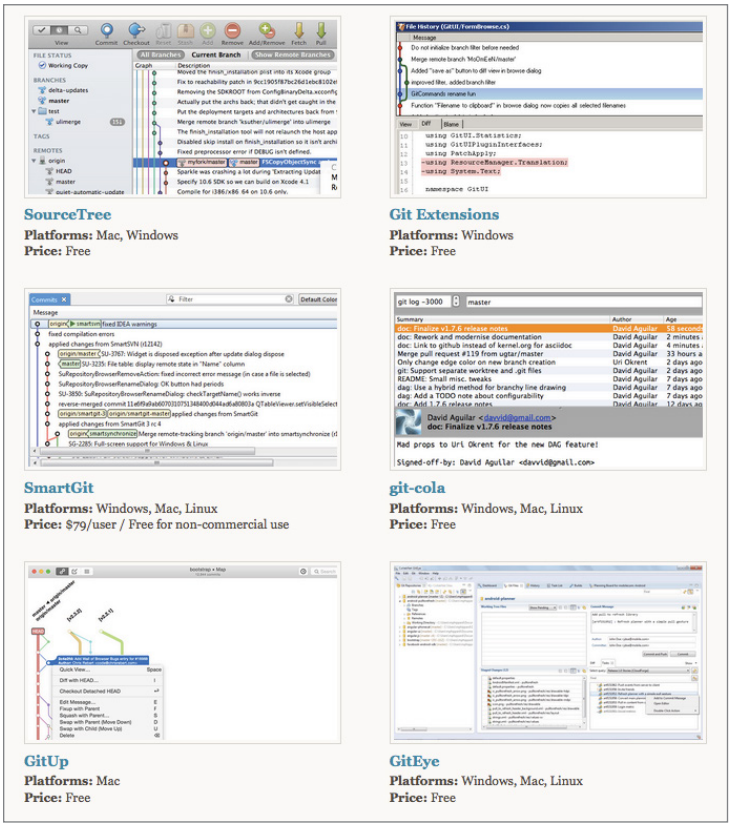


FIGURE 1-3: Examples of GUIs available for Git (from git-scm.org)

Plug-ins

Plug-ins are software components that add interfaces for working with Git to existing applications. Common plug-ins that users may deal with are those for popular IDEs such as Eclipse, IntelliJ, or Visual Studio, or those that integrate with workflow tools such as Jenkins or TeamCity. It is now becoming more common for applications to include a Git plug-in by default, or, in some cases, to just build it in directly.

Tools That Incorporate Git

Over the past few years, tooling has emerged that directly incorporates and uses Git as part of its model. One example is Gerrit, a tool designed primarily to do code reviews on changes targeted for Git remote repositories. At its core, Gerrit manages Git repositories and inserts itself into the Git workflow. It wraps Git repositories in a project structure with access controls, a code review workflow and tooling, and the ability to configure other validations and checks on the code. Figure 1-4 shows an example of a Gerrit screen.

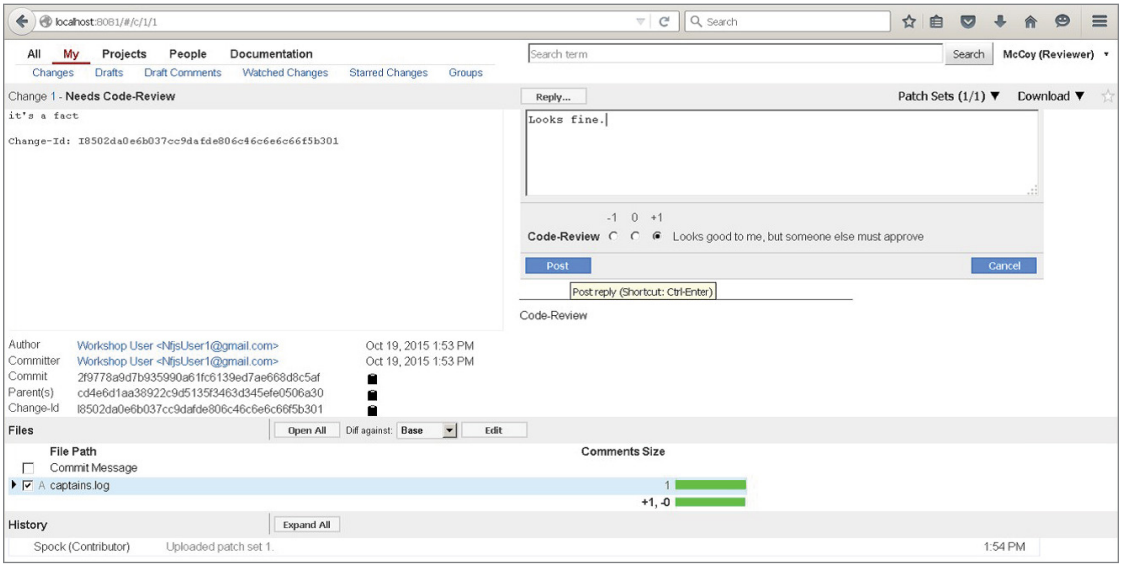


FIGURE 1-4: Example Gerrit screen

Git Libraries

For interfacing with some programming languages, developers have implemented libraries that wrap those languages or re-implement the Git functionality. One of the best-known examples of this is JGit. JGit is a Java library that re-implements Git and is used by a number of applications such as Gerrit (mentioned in the previous section). These implementations make interfacing with Git programmatically much more direct. However, there is sometimes a cost in terms of waiting, when new

features or bug fixes that are implemented in the core Git tooling have to be re-implemented in these libraries.

GIT'S ADVANTAGES AND CHALLENGES

Everyone has opinions, and anyone who's tried Git has an opinion about it. These usually vary from believing it's the greatest thing since sliced bread to wondering how they could ever effectively use it. In this section, you'll look at some of the advantages and challenges that Git offers (in no particular order). Granted, these lists are subjective, but themes in each area seem to consistently emerge.

The Advantages

Git is popular for many reasons. There are some things it just does better (faster, easier) than other source management systems and some things that it takes a totally different approach on. Learning about and leveraging the aspects outlined here will allow you to get the most out of this tool.

Disconnected Development

The Git model provides a local environment where you can work with a local copy of a server-side repository (this server-side repository is known as the *remote* in Git terminology). This copy resides within your workspace. When you are satisfied with your changes in this local repository, you then sync the local repository's contents up with the remote side.

All of the source management commands that you need to make changes can be run in this local environment. There's no need to access the remote repository until you're ready to sync content. Because of this, you do not need a connection to the remote repository to conduct source management. You just work against the local copy.

Because you can perform source management tasks in your local environment without needing a connection to the remote-server side, you can work *disconnected* from the remote and even disconnected from a network. This is what *disconnected development* means.

One important factor to keep in mind is that until you sync up with the remote, all of your changes and data are only in the local environment on your system. This is usually the local disk on your machine.

Fast Performance

Git stores a lot of information. (I'll describe its internal storage model in the next chapter.) However, it is efficient both in the way it stores content and in the way it retrieves it. Internally, Git packs together similar objects. Externally, it uses a good compression model to send significant amounts of data efficiently through a network. Of course, this network performance may be mitigated by limiting factors such as network latency, but as a general rule, wait times for Git operations from the server are not a factor.

For changes in the local environment, Git is as fast as its commands can be executed on your disk. Because it only has to interact with a local repository (in most cases not going across a network connection), the performance is equivalent to operating system commands.

Another factor that aids Git's performance is that it is designed to manage multiple smaller repositories—rather than larger aggregate ones that may be present in traditional source control systems. For example, consider how you might store the source code for a large Java project. In a traditional source control management (SCM) system, you might have a single large Java repository with all of the source code in subdirectories for the different JARs. However, in Git you would typically have a separate repository for the source code for each JAR. This granularity contributes to the smaller amount of content that has to be moved around in Git, and thus to a faster operation.

Finally, branching is extremely fast in Git. I'll explain why in Chapter 8, but essentially, as fast as you can create a file on your OS, you can create a branch in Git. This means there is no more waiting for extended periods while the source management system branches your content. Deleting branches is just as quick. Merging is generally quick as well, assuming there are no conflicts.

Ease of Use

There's a paradigm shift that is required when learning to use Git. And a prerequisite to thinking that Git is easy to use is understanding it. However, once you grasp the concepts and start to use this tool regularly, it becomes both easy to use and powerful. There are simple default forms of commands and options. As your proficiency grows, there are extended forms that can allow you to do nearly anything you need to do with your content. In addition, almost everything about Git settings is configurable so that you can customize your working environment. (Git configuration is discussed in detail in Chapter 4.)

The primary mistake that most new Git users make is trying to use it in the same way that they've always used their traditional source management system. Usually this means that they are trying to map commands and workflow concepts from the previous system to Git's commands. However, trying to adhere too strictly to this approach with Git will actually make the learning curve steeper. A better approach is to consider what sort of source management outcome is needed (files in the repository, viewing history, and so on), and then take the time to learn how that workflow is done with Git. (The Connected Labs included throughout this book will aid this process significantly by providing hands-on experience with Git.)

SHA1s

The strange-looking name SHA1 is an acronym for Secure Hashing Algorithm 1. In short, it's a checksum. (It has its roots in the MD5 implementation if you're familiar with that.) Git computes SHA1s internally as keys for everything it stores in its repositories. This means that every change in Git has a unique identifier and that it's not possible to change content that Git manages without Git knowing about it—because the checksum would change. In Git, SHA1s represent a direct way to identify and specify the exact change that you want to work with.

Ability to Rewrite History

One aspect of Git that is different from most other source management systems is the ability to rewrite or *redo* previous versions of content stored in the repository—that is, *history*. Git provides

functionality that allows you to traverse previous versions, edit and update them, and place the updated versions back in the same sequence of changes stored in the repository. This is a powerful feature of the tool, but it can also be dangerous (see the section, "The Challenges: Ability to Rewrite History," later in this chapter).

When content that you're working on in your local environment hasn't yet been synched to the remote side, this is a safe operation. And when you need it, it can be very beneficial. For example, consider a case where you forget to include a file with a change, or even just need to do something as simple as modify the message associated with the change. Git provides an *amend* option that allows you to update or replace the last change made in the local repository.

Additional functionality makes it possible to take selected changes from one branch and incorporate them directly into the line of changes in another branch. Beyond that are levels of functionality for doing editing throughout the history of one or more branches. An example case would be removing a hard-coded password that was accidentally introduced into the history months ago from all affected versions.

Staging Area

Git includes an intermediate level between the directory where content is created and edited, and the repository where content is committed. New users typically don't see this extra level as a positive, due to the perceived inconvenience of having to move content through another level. However, it does provide a separate area for use in some of Git's advanced operations, such as the amend option discussed previously. It also simplifies some status tracking. I'll cover the staging area in detail in Chapter 3.

Strong Support for Branching

Using branches is a core concept of Git. Earlier, I mentioned the speed with which users can create, delete, and manipulate branches. However, beyond that, Git provides capabilities for changing branch points and reproducing changes from one branch onto another branch—a feature referred to as *rebasing*. This ease in working with and manipulating branches forms the basis for a development model with Git. In this model, branches are managed as easily as files are in some other systems. Later in the book, I devote entire chapters to branching concepts.

One Working Area, Many Branches

It is rare these days for source management users to only be concerned with one release of content. Even when products are managed via a continuous delivery process, in a user's local environment, there are typically multiple changes underway, for new features, bug fixes, and so on. Traditionally, the best way to develop these multiple changes in parallel has been in separate workspaces, and, depending on the scope and ease of use of the source management application, in separate branches. With legacy SCM systems, maintaining these multiple workspaces, switching contexts between them, and ensuring they are up to date with the correct source code is a multi-step process that requires tracking and coordination by the user.

In Git, this is a single-step process managed by Git. Git allows you to work in one workspace for a repository, regardless of how many branches you may have or need to use. It manages updating the content in the workspace to ensure it is consistent with whichever branch is active. You never need

to leave that workspace. Also, while working in one branch, you still have the expected access to view, merge, or create other branches.

WORKING IN MULTIPLE BRANCHES SIMULTANEOUSLY WITH GIT

If you do find yourself needing to work in multiple branches at the same time, recent versions of Git have introduced a new feature to support this—worktrees (otherwise known as working trees). Worktrees provide a way to have and use multiple working directories with different branches (at the same time) all tied back to the same local Git repository.

We discuss worktrees in detail in Chapter 14.

The Challenges

Now, to balance out the picture, let's look at a few of the things about Git that can be challenging—especially for new users. I'll have more to say about this topic, including what to watch out for, and strategies for effectively dealing with these challenges, throughout the book.

Very Different Model from Some Traditional Systems

Going from a more traditional, centralized version control system to a distributed version control system such as Git requires a change in how you think about your source management workflow. Git implements a local environment with multiple levels in addition to a separate *remote* repository. As well, it operates with units that map more closely to directory tree structures than just individual files. This leads to considerations when creating and working in Git repositories, in terms of size and scope, that you don't usually worry about with centralized systems.

Different Commands for Moving Content

In most traditional source control systems, there are one or two commands for getting content out (checkout) and one or two for putting content in (check-in, commit), with options for modifying their behavior to work in different ways if needed.

With Git, there are different commands for moving content between the different layers, and these commands must be used in a particular sequence. This isn't really an issue after you've been working with Git for a while, and actually is clearer when talking about the workflow. However, it can be a little confusing to new users.

Staging Area

As previously mentioned, Git includes a staging level. This is an intermediate area that new code has to travel through on its way to the local repository. This will seem cumbersome at first, because content must flow through it, even in some situations where it doesn't appear to add value.

However, once you are comfortable with it, it will allow you to work with a power and flexibility that you haven't experienced previously.

Mind Shift and Learning Curve

All of the things I'm talking about as advantages and challenges contribute to the power of Git—as well as the learning curve. As I alluded to previously, one of the fundamental mistakes that new Git users make is trying to map too many concepts and workflows that they've used in the past with other systems, too closely to Git concepts and workflows. They often expect a one-to-one fit, just with different names. The basic principles of source management still apply—tracking changes, putting code in, getting code out, and so on. However, Git adds layers of flexibility and power on top of those principles, at the cost of requiring you to think differently about the units and stages of source control.

This requires a learning curve and a willingness to accept some features and requirements as useful, even if they don't immediately appear so. It's one of those situations where a feature won't seem beneficial until it is. As you continue to use the tool, it's a pleasant experience when you encounter those situations where you need to do X, you wonder if Git can do X, and you discover (in most cases) it can. Of course, there's also a learning curve with figuring out the exact invocation, and implications, of doing X.

Part of the mind shift comes early on in thinking about what should be in your Git repositories and branches. Just converting existing repositories one-to-one from another source management system is seldom the best approach. This is due to the way that Git manages scope in terms of changes and repositories. I'll discuss more about this as you learn more about Git.

Finally, it's worth pointing out that Git offers a built-in way to learn and explore the tool and workflow as you're going through this mind shift and learning curve—the local environment. I'll talk more about this in the next couple of chapters, but for now, know that you have the ability to make any source management changes (and mistakes) you need to in your local environment before you ever push them over to the remote environment, where others can see or access them.

Limited Support for Binary Files

Most source management systems do not have strong support for binary files, and Git is no exception. There are two aspects of dealing with binary files that are challenging here: internal format and size.

Because of the internal format of these types of files where the bits rather than the characters are what is important, standard source management operations can be difficult to apply or may not make sense at all. An example of the former would be diffing. An example of the latter would be managing line endings. If the SCM does not recognize or understand that a particular file is binary and tries to execute these types of operations against it, the results can be confusing and problematic.

The size of binary files can routinely be much larger than text ones. Very large binary files can pose a challenge for a system like Git since they usually cannot be compressed very much, and so can

impose more time and space to manage, leading to extended operation times when the system has to pass around these files such as when copying to a local system.

Of course, larger text files can also pose size challenges, but with text files, the ability to compute differences between versions and more compressibility can work better with Git's internal strategies for efficiently storing and serving these files.

Git has built-in mechanisms for identifying files as binary. However, it is also possible (and a best practice) to use one of its supporting files—the Git Attributes file—to explicitly identify which types of files are binary. Git Attribute files are covered in detail in Chapter 10.

The challenges with large binary files for source management in general have led to the development of several separate applications to help. Artifact repositories, such as Artifactory and Nexxus, are targeted specifically at storing and managing revisions of binary files. And the Git community itself has created various applications targeted at helping with this. Currently, the best-known one is probably Git LFS (Git Large File Storage)—a solution from the Git hosting site, GitHub. This application stores large files in a separate repository and stores text pointers in the traditional Git repository to those large files.

No Version Numbers

As referenced in the previous section on SHA1s, Git creates checksums (SHA1s) for everything that it stores. From one perspective, the overall SHA1 value for a change can function like the version number in most other source control systems. However, unlike traditional version or revision numbers, these are not short, easily remembered identifiers. SHA1s are actually 40-character hexadecimal strings. So, from a user perspective, SHA1s are not as convenient to remember, find, or communicate about. Typing one also requires some care.

Fortunately, in any Git instance, you only need to use enough of the characters from any SHA1 to uniquely identify that SHA1 from any other—usually the first seven characters. You can also use other references, such as tags or branch names, to indicate revisions where appropriate.

Merging Scope

While talking about the Git model, I mentioned that Git *thinks* in units that more closely map directory structures than individual files. This difference in granularity provides advantages in managing and manipulating changes in source control. However, it can also create disadvantages in merge situations where there are conflicts. Simply put, any two changes by different users within the scope of a *commit* can be a conflict, even if they are in entirely different files or directories. As a result, the more people that are making changes within the scope of a repository, the more likely they are to encounter merge conflicts when trying to get their updates in. This is a factor to consider when planning how to structure your Git repositories.

Ability to Rewrite History

Git's ability to rewrite history falls into both categories. On the challenging side of the scale is the potential impact that uncoordinated use can have on other users. Suppose that multiple users have obtained content from a remote (shared) Git repository. One user decides to perform an operation that changes the revision history. Changing the history results in new internal checksums (SHA1s) for changes in the repository, starting at whatever points the revisions were made. Once the updates

are put back on the remote side, any other users that need to merge in updates will have to deal not only with the newest content, but also with the changes to the revisions in the history made by the other user. At best, this can be surprising. At worst, it can be very time-consuming and resource-intensive, because it requires them to incorporate all of the changes.

As a highly recommended guideline, changes that alter history should only be made in a user's local environment *before* the affected revisions are pushed across to the remote side. If there is a critical need to change revisions in the history of a repository after it has been made available on the remote side, then there is a recommended approach: other users should be informed in advance, and given a chance to get their changes in before the changes to the history are made. After the changes are completed, they can get a fresh copy to work with locally. This will allow them to avoid potentially difficult merge situations.

Timestamps

When using most source control systems, timestamps that reflect when changes were made in the repositories are a useful and static property. Given any point in time, it is possible to pull the content from the repository as it was at that point and always get the same set of content on subsequent pulls. Not so with Git. Due to the way that remote repositories are synched from local repositories, the timestamp that shows up in the remote repository is the time the update was made on the *local* environment, not the timestamp of when things were synched to the remote.

This means that it's possible to pull content from the remote side based on a particular timestamp and get a certain set of content, then later pull it again based on the same timestamp, and get a different set of content. This can happen if one or more changes were made in a user's local environment, prior to that timestamp, but weren't synched to the remote until between the two pulls.

In this case, a new change with an older timestamp would suddenly show up in the remote. For this reason, you can't rely on timestamps for some of the cases where they are traditionally employed with existing source control systems. I will discuss what the alternative is for Git when I talk more about the remote side in Chapter 12.

Access and Permissions

Out of the box, Git does not provide a layer to set up users or to grant and deny access. For the local environment, this doesn't matter because everything is, well, local. For shared, server-side repositories, there are a few options:

- Using operating system mechanisms such as groups and umasks that limit the set of users and their direct repository permissions
- Limiting access via client-server protocols (SSH, HTTPS)
- Adding an external applications layer that implements a more fine-grained permissions model and interface

Note that these are not mutually exclusive. In a corporate environment that chooses to host its own shared, server-side repositories, for example, you would want to limit who could directly access the actual repositories on disk at the system level, have authentication for users who need to put content into them from their local environments, and potentially have a permissions layer that can be centrally managed or managed by a team within a selected scope.

SUMMARY

In this chapter, I introduced Git, discussed where it came from, and talked about some of the advantages and disadvantages that users should be aware of when working with this tool. Along the way, I also introduced a number of terms and concepts that are part of Git. In subsequent chapters, I will be expanding on and explaining what each of these terms and concepts means, along with teaching you how to use them.

If you're coming from an environment where you used a traditional centralized source control system, you'll find that Git is significantly different and has a learning curve. The workflow is different as well. Trying to map commands, structures, and workflows from your previous system is not an effective strategy. Rather, you should take the time to read through the following chapters and examine the concepts and examples. Equally important is that if you can work through the Connected Labs, they will go a long way toward helping you internalize the concepts, ensure a deeper understanding of the material, and help you be ready to apply Git to your job when you need it.

In Chapter 2, you'll look at some of the primary design concepts that Git uses internally and that are helpful for users to understand before going further with it.

2

Key Concepts

WHAT'S IN THIS CHAPTER?

- The differences between a centralized and distributed source management system
- The differences between a traditional *delta* model for tracking source code changes and the way that Git tracks changes
- Why Git is efficient
- How (and why) Git repositories should be organized
- Things to keep in mind when migrating repositories to Git
- Dealing with large files in Git

In this chapter, I'll explain some of the underlying key design concepts that Git uses. Implementation around these concepts forms the basis for how Git works and how to use it. I'll broadly break these concepts down into two categories, user-facing and internal, and show how they differ from more traditional source management systems. Lastly, I'll focus on some important considerations for creating repositories in Git, and managing special content such as binary files.

DESIGN CONCEPTS: USER-FACING

Version control systems (VCS) such as Git can be broadly classified as either *centralized* or *distributed*. Git is an example of a distributed version control system (DVCS). Other systems in this category include Mercurial and Bazaar. Examples of a centralized version control system (CVCS) would be Concurrent Versions System (CVS) and Subversion.

The fundamental differences between a DVCS and a CVCS have to do with how the system manages repositories and the workflow that the user employs to get content into the server-side part of the system.

Centralized Model

Figure 2-1 illustrates a traditional centralized model. In this model, you have a central server that holds all of the repositories with all of the history and all versions of changes that have been put into the system over time. This area is the one source of the truth—the container of all the repositories.

When users want to work with a file in one of these repositories, they connect to the server via a client, and retrieve the files and the versions they want to work with. They then make whatever changes they need to, connect to the server again, and send the update back to it. There, the differences from the previous version are determined and stored in the repository as updates.

In this type of model, users are dependent on the central server. If, for some reason, users cannot connect to the server, they cannot do any source management operations.

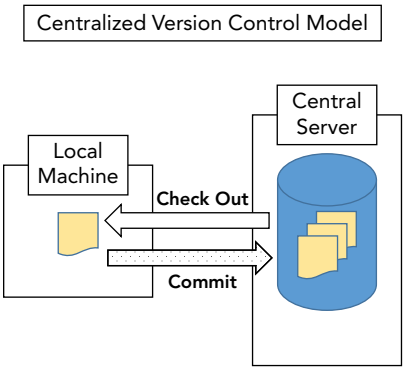


FIGURE 2-1: A traditional centralized version control model

Distributed Model

In a distributed system, the model is somewhat different. There is still a server that holds the shared repositories, and that clients interact with. However, when users want to start making changes, instead of getting individual files or directories from the server, they get a copy of the entire repository. The copy comes from the server side and has all content (including history) up to the point in time when the copy is created.

In Git terminology, the server side is called the *remote repository* (or just remote). The copy operation is referred to as a *clone*. You can call the area on your local system with the cloned repository your *local environment* because it consists of several layers (which you'll explore in the next chapter). For simplicity, I'll refer to the remote repository as just the *remote* throughout the rest of this discussion. Figure 2-2 illustrates this model.

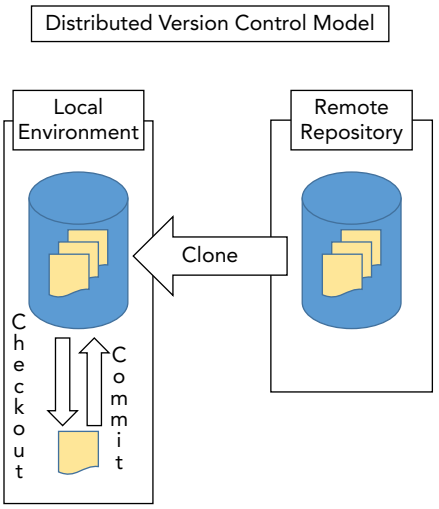


FIGURE 2-2: A distributed version control model

The actual cloned (copied) repository within the local environment is called the *local repository*. It has all of the files, histories, and other data that were in the remote. A change that is made into the local repository is called a *commit*, similar in concept to a *check-in* in some other systems.

Once users have cloned from a remote, they can do all of their source management operations against the local repository. When users have made all the commits they want in the local repository, they then push their changes to the remote.

The key difference here is that, in a DVCS such as Git, users are performing the source management operations against a local copy of the server-side (remote) repository instead of making them against the actual server-side repository. Until users need to push the changes back to the remote, they do not even need to be connected to it. The connection between the local and the remote side is not constant. Rather, it is activated when updates need to be synchronized between the two repositories.

Because users do not have to be connected to the remote to do their source management operations, they can work disconnected from the remote. As noted in Chapter 1, this is referred to as being able to do *disconnected development*. Figure 2-3 shows a conceptual model of this approach.

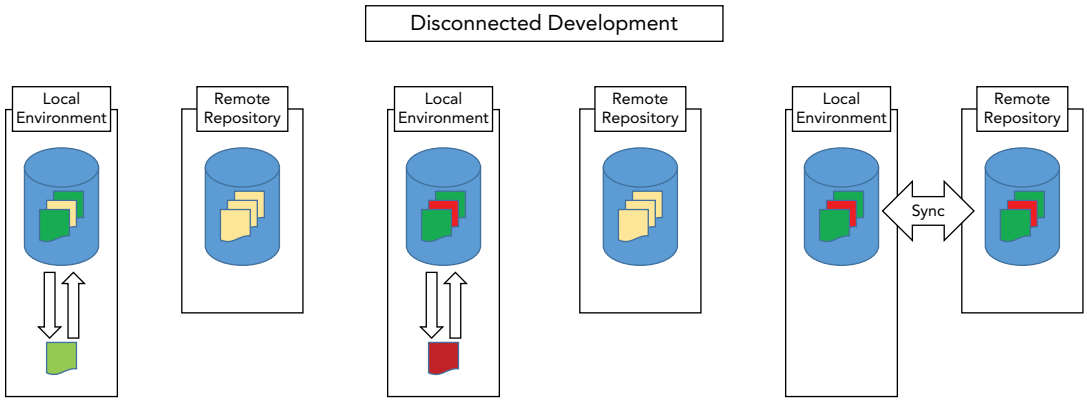


FIGURE 2-3: Disconnected development

In Figure 2-3, starting on the left, a user makes a change to a file in the local repository without any connection to the remote. Then a second change is made in the same way. Finally, the local environment is synched up with the remote side so that both areas have the latest content.

One other thing to note is that a remote can actually be any Git repository that is set up to function that way. Most commonly, a remote is a Git repository hosted on a server and running as a daemon process. However, there are various protocols for communicating between Git clients and servers, even a simple one that operates via shared folders. I'll have more to say about these protocols in Chapter 12 where I discuss remotes in more detail.

DESIGN CONCEPTS: INTERNAL

Another area where Git differs significantly from traditional source management systems is in the way it represents and stores changes internally.

Delta Storage

In a traditional source management system, content is managed on a file-by-file basis. That is, each file is managed as an independent entity in the repository. When a set of files is added to a repository for the first time, each file is stored as a separate object in the repository, with its complete contents. The next time any changes to any of these files are checked in, the system computes the differences between the new version and the previous version for each file. It constructs a delta, or *patch set*, for each file from the differences. It then stores that delta as the file's next revision.

This model is called *delta storage*. Figure 2-4 illustrates this process.

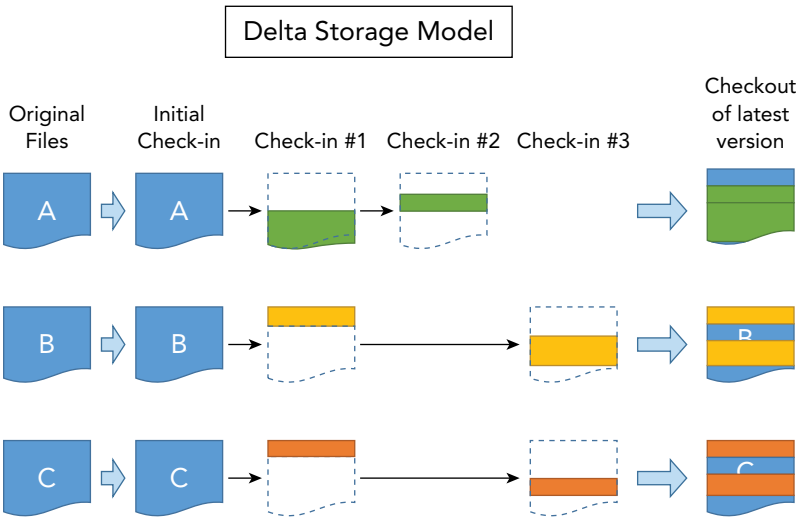


FIGURE 2-4: The delta storage model

In the first iteration, files A, B, and C are checked in. Then, changes are made to the three files and those changes are checked in. When that occurs, the system computes the deltas between the current and previous versions. It then constructs the patch set that will allow it to re-create the current version from the previous version (the set of lines added, deleted, changed, and so on). That patch set is stored as the next revision in the sequence. The process repeats as more changes are made. Each delta is dependent on the previous one in order to construct that version of the file.

In order to get the most current version of a file from the system when the client requests it, the system starts with the original version of the file and then applies each delta in turn to arrive at the

desired version. As the files continue to be updated over time, more and more deltas are created. In turn, more deltas must be applied in sequence to deliver a requested version. Eventually, this can lead to performance degradation, among other issues.

Snapshot Storage

Git uses a different storage model, called *snapshot storage*. Whereas in the delta model, revisions are tracked on a file-by-file basis, Git tracks revisions at the level of a directory tree. You can think of each *revision* within a Git repository as being a slice of a directory tree structure at a point in time—a *snapshot*. The structure that Git bases this on is the directory structure in your workspace (minus any files or directories that Git is told to ignore—more about that later).

When a commit is made into a Git repository, it represents a snapshot of part or all of the directory tree in the workspace, at that point in time. When the next commit is made, another snapshot is taken of the workspace, and so on. In each of these snapshots, Git is capturing the contents of all of the involved files and directories as they are in your workspace at that point in time. It's recording the full content, not computing deltas. There is no work to compute differences at that point.

The snapshot storage model is shown in Figure 2-5. In this model, you have the same set of three files, A, B, and C. At the point they are initially put into the repository, a snapshot of their state in the workspace is taken and that snapshot (with each of the file's full contents) is stored in Git and referenced as a unit.

As additional changes are made to any of the files and further commits are done, each commit is built as a snapshot of the structure as it is at that point. If a file hasn't changed from one commit to the next, Git is smart enough not to store a new version, and just creates a link to the previous version. Note that there are not any deltas being computed at this point and you are managing content for the user at the level of a commit rather than individual files.

Later, when you want to get one of these snapshots back, Git can just hand back the specific set of content associated with that commit, without going through the extensive reconstruction process required by the delta model.

Git's Storage Requirements

One of the questions that usually comes to mind right away when people are introduced to the snapshot storage concept is, "Doesn't this use a lot of disk space?" There are a couple of points related to that. First, as I just noted, Git can use links in some cases to reduce duplicate content. Second, Git compresses content using zlib compression. (Notice the smaller compressed size of the blocks representing content in the repository in Figure 2-5.) Third, periodically, at certain trigger points, such as when running garbage collection functionality, Git looks for content that is very similar between revisions and packs those revisions together to form a compressed *pack file*. In these cases, it can actually create an associated *delta* of sorts that represents the differences between very similar revisions. The delta here is what it takes to get back to previous revisions. Git assumes that the most recent revision is the one that will be most requested and thus best to keep as a full, ready revision.

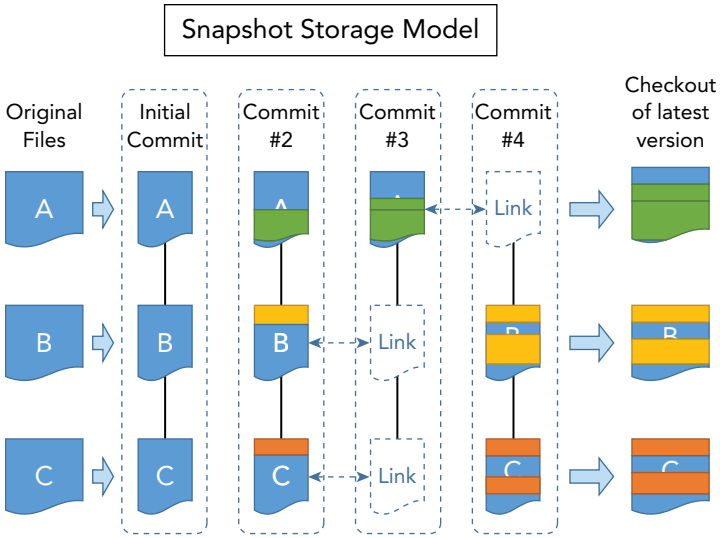


FIGURE 2-5: The snapshot storage model

So, in the Git model, the use of any deltas is a deliberate optimization for storage rather than the default versioning mechanism. Figure 2-6 illustrates a way to think about this concept, where multiple objects have been packed together internally. This is invisible to the user. From a user perspective, Git still manages interactions with the user in terms of individual snapshots, regardless of whether or not content ends up packed in the repository.

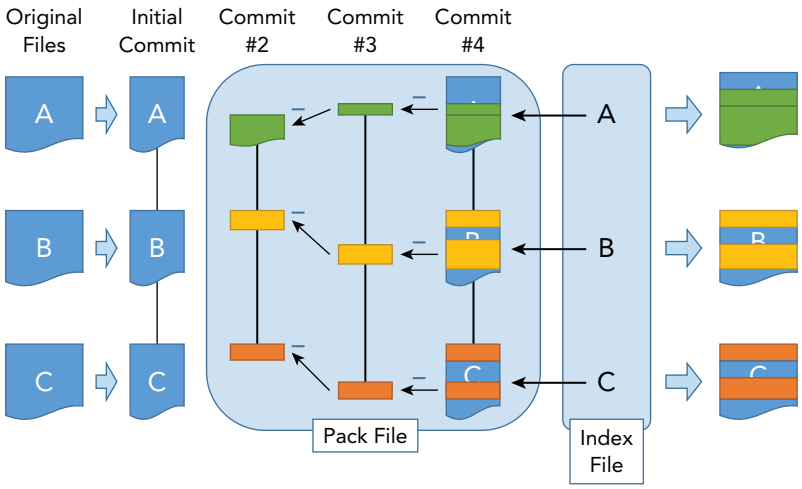


FIGURE 2-6: A representation of Git's packing behavior to optimize content size

All of these approaches help to reduce the space a Git repository requires. In fact, if you were to compare the corresponding disk space requirements for a source control system that uses the delta model to the snapshot model that Git uses, you might find that in the best cases, Git actually uses less.

(You may be wondering how a model like this handles binary files since those don't lend themselves to a delta model. I cover dealing with Git and binary files in more detail later in this chapter.)

A final, related point is that Git is designed to work with multiple, smaller repositories rather than large, monolithic repositories, a characteristic I'll explore in more detail in the next section.

So, to summarize, there are two differences between delta and snapshot storage:

1. Delta storage manages content on a file-by-file basis, as opposed to snapshot storage where content is managed at a directory tree level.
2. Delta storage manages versions over time by figuring out the differences and storing that information from revision to revision (the delta). It reconstructs later revisions by starting with the base version and applying deltas on top of that. Because snapshot storage is storing a capture of the entire tree, it does not usually have to do any reconstruction, or only a very small amount if the content has been packed.

Git's approaches in these areas create a very powerful model to build on, especially as they pertain to branching. However, they also create the need to structure repositories appropriately in Git for the best usability and performance. This is the topic of the next section.

REPOSITORY DESIGN CONSIDERATIONS

When beginning to work with Git, whether creating repositories for new content or migrating existing content from another source management system, it is important to consider how you size and structure your repositories. For existing content, unless your code is already broken down into very distinct, separate modules, a one-to-one migration is unlikely to be the best approach. This is because of repository scope.

Repository Scope

A key point to keep in mind when beginning to work with Git is that it is designed to be used as a set of many, smaller repositories. How small? Well, as an example, consider the case of a Java project managed in a traditional, centralized source management system. You might have a single repository for a Java project that's made up of ten different JARs, with all of the source code for all of the JARs stored in different subdirectories in the repository. This arrangement typically works well in a centralized model where each file is managed separately. In the working model for that system, you don't typically check out or check in the entire repository each time. You can manage things at smaller granularities, such as only checking out the subdirectory with the code for one particular JAR, modifying a few files, and then checking those files back in.

In the Git model, a more common scenario would be to have a separate repository for the code associated with each separate JAR. Why? Recall that Git manages changes as commits that are a snapshot of the larger workspace—the set of files and directories. While Git is efficient in how it stores and retrieves data, this efficiency is still relative to the size of the content. If the content is inordinately large, you may find yourself waiting longer than you'd expect for operations that get or put data from or into the repository.

In addition, as I alluded to in Chapter 1, because Git manages content in terms of snapshots, any changes by two users within the scope of the same snapshot, *regardless of whether or not they are to the same file*, have potential to cause a merge conflict, depending on timing.

To illustrate this, suppose you and another user clone the same repository in Git down to your local systems, and the repository contains directories 1 and 2. The other user makes a change in file A in directory 1, commits it, and pushes it up to the remote. Then you make a change in file B in directory 2, and commit and attempt to push your changes back to the remote. Git will reject your changes at the point where you try to get them into the remote. This is because Git considers that something else (anything else) has changed in this repository since you originally got your copy of the code. Even though you didn't touch the same file as the other user, you have a merge conflict within the snapshot, because someone else made a change before you could get yours in. This is one of the key frustrations for new Git users. I'll talk more about this in Chapter 13, including how to resolve the merge conflicts. (Also, see the following Note.)

In addition to repository size, there's a second point to consider. Ideally, you want to create repositories that will not have too many users working in them at the same time, and making (from Git's viewpoint) conflicting changes. This will help limit the number of rejected pushes and the amount of merging work that has to be done.

> **NOTE** *To be fair, resolving these kinds of conflicts is generally an easy mechanical process, unless both users have changed the same file or files. However, it does involve additional operations and inspection by the last user who is trying to get their changes in. Depending on the scope, the time to review the conflicts can be non-trivial.*

Having smaller repositories with only a few users making changes also allows for closer collaboration and coordination. It helps to keep operations in Git working quickly and smoothly when Git is manipulating content at the scope of a repository. This also applies to development environments, such as Eclipse, that look at projects as equating to a repository when interfacing with Git.

In general, you can think of one repository in Git as equating in scope to one module of your project. If your code is not already modularized, it can sometimes be difficult to figure out what should constitute a module. One general guideline is to map the code to build a JAR, DLL, EXE, or other single component to a repository. Think in terms of what code you would use to build a single deliverable in an application such as a Gradle or Maven project or a developer interface such as Eclipse, IntelliJ, or Visual Studio. Consider code that is owned and maintained by only one or a few people

to reduce the risk of merge conflicts. If your code does not easily map out this way, then it's worth spending some time up front to figure out how to get it into a structure that is more modular. You can then base your Git repositories on that revised structure.

When considering how to organize code in Git repositories, it's also important to consider whether all categories of content related to a module are appropriate to migrate or store in a repository. There are general guidelines (especially around very large files) that apply, mostly independent of the source management application. I'll explore those guidelines next.

File Scope

When dealing with very large files, there are a number of considerations and approaches to take into account. An arbitrary definition of *very large* might be over 100 MB for text files, but less for binary files for reasons I'll talk about in the next few sections. Nearly all of these considerations apply to any source management system, not just Git. I'll now discuss some points you should consider.

Storage Model

Source management systems can't create deltas between versions of binary files. As a result, they end up storing full versions for each change to a binary file. This is necessary, but inefficient, and can quickly consume significant disk space if the files are large. Even in a system such as Git that compresses content, most binary files do not compress well. For certain types of smaller binary content, such as icons or other graphical elements, storing those files in the system usually doesn't present a problem and makes sense. For larger files, some pre-planning of alternative approaches to managing these files can help avoid issues in the repository. One common alternative approach for dealing with these files is to store them in a separate repository.

Separate Repositories

For the reasons outlined previously, storing very large files, especially binaries, in a repository such as Git is not the best approach. This also applies to generated files. Instead, there are specially designed applications for working with these types of files: *artifact repositories*. Artifact repositories work much like a source control system, but are designed to be a good fit for managing versions of files that don't really belong or fit well in your standard source repositories. Builds and other parts of a pipeline can pull source code from the source management system and resolve needed pre-built binary dependencies from artifact repositories. Some of the more popular artifact repositories today include Artifactory and Nexus.

There is also an option to store large files that need to be managed in source control in a second, separate Git repository designated for them. This approach still suffers from the problems discussed in the "Storage Model" section. However, it does remove the impact of dealing with the large binaries in the other smaller repositories.

Extensions to Git

Not surprisingly, a set of applications and packages has been created around trying to solve the limitations of Git with large files. Among these are extensions to Git, such as the git-annex and Git Large File Storage (Git LFS) open-source packages. There are also other packages, but these two

seem the most likely to continue to receive support and development. This is primarily due to their incorporation into two of the major Git-hosting applications: git-annex has now been incorporated into GitLab as GitLab-Annex, and Git LFS is now incorporated into GitHub as well as some versions of Bitbucket—another Git repository hosting system.

In these implementations, the large files are stored in a separate space, but referenced by pointers inside of a normal Git repository. The applications vary in terms of characteristics that include the following:

- Performance
- Configurability (Can files be stored on user-configurable locations?)
- Ease of use (registering of files and use of existing commands versus new commands)
- Cost for long-term/large-scale use
- Learning curve

All of these characteristics factor into the transparency and usability of the process, but some setup and overhead is always required.

Generated Content

Files generated from source code stored in your source control system should not actually be stored in the source management system. If these files are generated from sources that you have control over, then the file can always be reproduced from the sources.

In a model where the generated files are stored in the source repository, if the sources change frequently, then the generated content must also be updated frequently in the repository. This can be challenging to keep in sync and can lead to the problems discussed in the "Storage Model" section.

Generally, the reason why files produced from existing source are stored in the source management system boils down to having them easily accessible or using the source management system as a transport mechanism between processes. However, there are better ways to manage those needs such as using an artifact repository (described in the "Separate Repositories" section) that is designed for this purpose.

MANAGING BINARY FILES IN GIT

While I am talking about binary files, it's worth discussing how Git identifies and manages these files. Git can read a separate configuration file called a *Git Attributes* file (named .gitattributes on disk) to determine how to treat certain file types. In this file, different file types can be identified as *binary*. For such types, Git understands that it should not perform some of the operations that it does with text files, such as diffing and modifying line endings.

I'll talk in detail about the Git Attributes file in Chapter 10.

Shared Code

While I'm on the topic of easily accessing code in the source management system, at times, it may seem that you need to share code from one repository to another. Git provides a way to do this through a construct called *submodules*. A submodule is essentially a static reference to another repository that resides in your local environment. Git understands that it is a separately managed repository even though it is in your tree structure.

Submodules can be useful in certain cases, such as when an organization needs to share source for development dependencies that are being worked on by one group with other groups. However, they can be challenging to keep in sync without careful attention to updates. Managing them requires a different, coordinated set of operations. And it can be easy to back-level them for yourself or other users. For these reasons, submodules can be problematic and are not generally recommended for beginning Git users.

Git also supports another construct called subtrees that provides similar benefits to submodules, but with a simpler structure and a simpler set of operations to manage them. Both submodules and subtrees are explored in detail in Chapter 14 and the reader is advised to read that before attempting to use either of these constructs.

Another alternative approach is to just build the needed artifacts from other repositories separately and specify them as compile-time or run-time dependencies to pull them in, if this fits with how your project is organized.

SUMMARY

In this chapter, you learned about some of the differences between Git's overall design and functioning, and that of more traditional centralized source management systems. I covered the model that Git uses to clone a repository and create a stand-alone local environment in which to do source management operations versus the typical legacy "always do the operations to the server" model. Along these lines, I talked about how Git is structured with the local environment and the remote environment. I also introduced the concept of disconnected development, which is one of the appealing aspects of using Git. All of this allows you to get things the way you want them locally before you share them back with others in a public repository.

I also shared some insights on how Git manages things internally. You learned how Git sees sets of files involved in a commit as a unit and works at a granularity that is directory tree–based, not file-based. You also looked at how it stores and manages commits over time.

Finally, I discussed some considerations when creating or migrating to Git repositories, defining some guidelines for repository scope and file scope, especially around large files and binaries. Git is not strong in managing very large files, but there are good alternatives.

In the next chapter, you'll expand your understanding of the local environment that Git provides by looking at the Git promotion model, as well as looking at the workflow to move content through the different levels.

3

The Git Promotion Model

WHAT'S IN THIS CHAPTER?

- The different levels of Git
- The workflow for moving content between the levels (the Git promotion model)
- Why Git has the staging area and how it is used
- A summary of the commands that you use to move content between the levels

Whenever you are learning a new system or process, it's convenient to think about it in terms of something you already know or have some familiarity with. In this chapter, you will take a tour of the various levels that make up a Git system. You will also relate them to a common model that almost everyone who works in an IT-related field will recognize. This model also provides a convenient way of thinking about how you get content through the levels, and introduces you to the basic Git commands for a workflow.

In addition, I'll focus in on one level that is not typically found in other source management systems, but which plays a key role when interacting with Git and some of its advanced functionality. Understanding this level early on is a prerequisite to really understanding any Git workflow.

THE LEVELS OF GIT

So far, I have introduced Git and discussed its history, good points, and not-so-good points. I've also presented some concepts to help you understand its internal functioning. It's now time to look at the different levels that users encounter when working with Git. These levels represent the stages that content moves through, as it makes it way from the local development directory to the server-side (remote) repository. One way to think about and understand these levels is to compare them to another well-known model, a dev-test-prod environment.

Dev-Test-Prod and Git

Figure 3-1 shows a simple block diagram representing a dev-test-prod environment. Most organizations employ some version of this model in their software development and release processes.

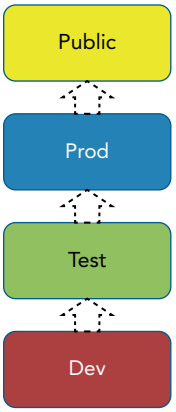


FIGURE 3-1: A simple dev-test-prod environment

You can think of this environment as a sort of *promotion model* where content moves up through the levels as it matures. Each movement can be initiated by someone or some process when it is deemed ready. At any point, different levels may contain the same or different versions of some particular piece of content, depending on which levels it has been promoted to and whether any additional changes have been made at a lower level.

To give you a better understanding of this environment, I'll briefly describe the purpose of each of the levels in my reference model.

At the bottom, you start with a Dev area (a development workspace) where content is created, edited, deleted, and so on. Some other names that might be used for this level include sandbox, playpen, workspace, and working directory.

When the code is deemed adequate, it can be moved to the Test area (the testing level). Not all of the code has to be moved to Test at the same time. This is an area where different pieces can be brought together to ensure that everything is ready for production.

Once a set of code has passed the testing phase, it can be promoted to the Prod (or production) area; this is where it is considered ready and officially released.

Then, for my purposes here, you add another level, Public, which represents an area where the production code is put, to be shared with others. An example might be a website where content is deployed so that others can see it and access it.

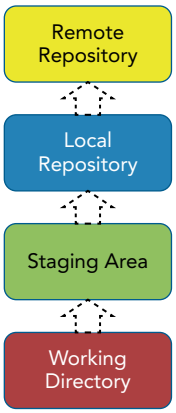


FIGURE 3-2: The levels of a Git system

Given this reference of a dev-test-prod(-public) model, let's look at the different levels that Git uses as an analogy to this model, and how they relate to each other. Figure 3-2 shows a similar way of thinking about the Git levels.

Starting at the bottom is the working directory where content is created, edited, deleted, and so on. Any new content must exist here before it can be put into (tracked by) Git. This serves the same purpose as the Dev area in the dev-test-prod-public model.

Next is the staging area. This serves as a holding area to accumulate and stage changes from the working directory before they are committed into the next level—the local repository. You can think of this process as being similar to how you might move content to the testing stage in your

dev-test-prod-public model. It is a place to *build up* a set of content to then promote. I'll go into more detail about this area shortly.

After the staging area comes the local repository. This is the actual source repository where content that Git manages is stored. Once content is committed to the local repository, it becomes a version in the repository and can be retrieved later.

The combination of the working directory, staging area, and local repository make up your *local environment*. These are the parts of the Git system that exist on your local machine—actually, within a special subdirectory of the root (top-level) directory of your working directory. This local environment exists for users to create and update content and get it in the form they want before making it available or visible to others, in the remote repository.

The remote repository is a separate Git repository intended to collect and *host* content pushed to it from one or more local repositories. Like the Public level in the dev-test-prod model, its main purpose is to be a place to share and access content from multiple users. There are various forms of hosting and protocols for access that I'll talk more about in Chapter 12. I'll refer to this as your *remote environment*.

Figure 3-3 adds the local versus remote environments encapsulation to the model. Let's examine each of these areas in more detail.

The Working Directory

Any directory or directory tree on your local system can be a working directory for a Git repository. A working directory can have any number of subdirectories that form an overall *workspace*. (You might also hear this referred to by similar names such as "working tree" or "worktree." In a tree structure, the higher-level directory where you initiated work with Git becomes the top level or root of your workspace. All subdirectories are considered part of the working directory's scope, unless Git is specifically told to ignore them via a .gitignore file (discussed in Chapter 10) or they are part of a Git *submodule* (discussed in Chapter 14).

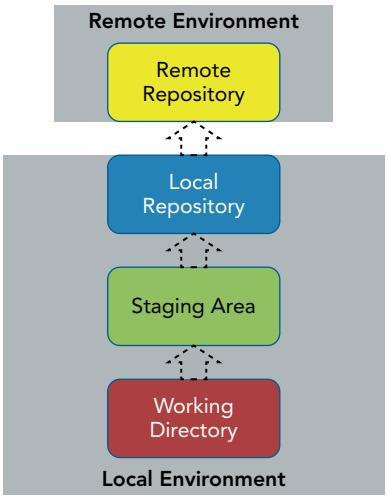


FIGURE 3-3: The local versus remote environments

When you connect Git to a local directory tree, by default Git creates a repository skeleton in a special subdirectory at the top level of the tree. That repository skeleton is the local repository. The physical subdirectory is named *.git* by default. This is a similar convention that many open source projects use, storing metadata in a directory starting with a period (.) followed by the name of the tool or application. Thus, your repository and all of your source management information is located within a subdirectory of your working directory.

OVERRIDING GIT'S DEFAULT LOCATIONS

In the section above, we noted that "by default" Git creates the repository skeleton under a subdirectory named .git at the top level of your source tree.

This is actually configurable through an option that we pass to Git (--git-dir) when running it, or through an environment variable ($GIT_DIR). However, unless you have a strong reason to change these, you are better off just leaving these as the locations set by default. Throughout the book, we will just refer to these settings as being ".git" under the working directory for simplicity.

As I discussed in Chapter 2, it's important to consider how much content you're trying to manage in any one Git repository, and thus in your working directory. Your repository structure, content, and scope are based on the structure, content, and scope of your workspace, and so similar guidelines apply. When developing code, a workspace should most likely consist of the structure needed to create a single deliverable—a JAR file or DLL, and so on. For other kinds of content, consider what makes sense as a logical unit that can be managed separately and maintained by a small number of users to reduce the occurrence of merge conflicts.

If you have content in your working directory that should not be tracked or managed by Git, then those files and directories should be listed in a .gitignore file at the top level of your tree. The .gitignore file is just a text file containing a list of files, directories, or regular expression patterns. Git understands that if this file exists, then Git should not add or track those files and directories listed in it. Common examples of types of files to have Git ignore would be very large files (especially binary files) and files that are generated from content already being tracked. (Refer to Chapter 2 for the reasons behind this.) The .gitignore file is discussed in detail in Chapter 10.

NOTE *Before going further, it's useful to clarify some Git terminology. People frequently talk about a* commit *in Git. In Git, a commit is both a noun and a verb, an entity and an action. Doing a commit (committing) means moving content from the staging area to the local repository. A commit as an entity means a set of content managed as a unit by Git. Similarly, the term* stage *or* staging *refers to the action of promoting content from the working directory to the staging area. I'll clarify all of this in the next chapter, but I'll need to use this terminology in talking about the remaining levels.*

The Staging Area

The staging area is one of the concepts in Git that many new users have difficulty understanding and appreciating. At first glance, it may seem like an unnecessary intermediate level that gets in the way of trying to promote content from the working directory to the local repository. In fact, it plays a significant role in several parts of Git's functionality.

What's the Point of the Staging Area?

As its name implies, the staging area provides a place to *stage* changes before they are committed (promoted) into the local repository. The staging area can hold any set of content that has been promoted from the working directory and is a candidate for going into the local repository—from a single file to all of the eligible files. The staging area provides a place to collect or assemble individual changes into the set of things that will be committed. It allows finer-grained control over the set of things that make up a change. Now let's look at the common use cases for it.

There are two ways of viewing the utility of the staging area: user-initiated and Git-initiated. Both ways offer benefits to the user; the difference is in which actions or processes place content into the level. You'll first look at the use cases or *scenarios* that originate with the user moving content into the staging area.

The *Prepare* Scenario

The first use case for the staging area can be thought of as the *Prepare* scenario. In this scenario, as a user completes changes in their workspace, they move files that are ready into the staging area. In the simplest case, this is a single promotion of all eligible content (any new or changed files that Git is not told to ignore). However, it can also be done at any granularity of files that the user chooses, meaning the user could even choose to promote each of the files one at a time into the staging area as work is completed.

Think of it like this: suppose you have a large checklist of files to modify in order to create a feature or fix a bug. As you complete changes on a subset of the files, you want to go ahead and promote that subset to ensure the changes are persisted, outside of your workspace, on your way to building up the full set for the change. As pieces of the larger change are done, you move those pieces to the staging area and check them off your list.

With other source management systems, you typically only have the workspace and the repository. And putting a subset that's an incomplete change into a repository can cause confusion, failed builds, and so on. That's because in those systems, committing changes means they go directly into a public/server-side repository where they are immediately visible and accessible by users and processes, rather than going into a local area first, as they do in Git. To avoid having those changes go directly into the public repository in other source management systems, you might resort to saving those changes off into another local directory—or just leaving everything in the workspace until you get the entire set of changes completed.

However, a more useful and elegant model would allow you to stage parts of changes outside of your workspace until you have a complete change built up and ready to commit into the repository. This is what Git allows you to do. Of course, there's no requirement to stage the change as separate pieces. You can promote everything as a unit from the working directory. However, as you become

more familiar with Git, and start to work with larger changes, you'll likely find more value in being able to break them up in this way. As well, Git allows for some interesting advanced functionality such as staging only selected changes from a file. You'll explore this workflow in more detail in Chapter 5.

The *Repair* Scenario

A second use case for the staging area can be referred to as the *Repair* scenario. In actuality, you might call it the *amend* scenario as it relies on an option by that name when doing a commit.

As I noted in the previous chapter, one of the interesting things that Git allows users to do is to *rewrite history*. That is, they can modify previous commits in the repository. The simplest way to do this is by using the amend option when doing a commit. This operation allows the user to pull back the last commit from the repository, update its contents, and put the updated commit back in place of the previous one. Effectively, it provides a *do-over*, or an opportunity to *repair* the last commit.

So where does the staging area come in for this mode? When the previous commit is amended, it is amended with any content that is in the staging area. The workflow is essentially as follows:

- Make any updates in the working directory.
- Put the updates into the staging area.
- Run the commit with the option to amend.

The last operation will cause the previous commit to be updated with whatever is in the staging area and then place the updated commit back into the local repository, overwriting the previous version. (If there are no updated contents in the staging area, then only the message that is attached to the commit can be updated.)

This is a powerful feature that gives users a lot of flexibility. As you may have gathered, one of Git's aims is to allow users to easily create and change things as many times as needed in their local environment before actually updating content (on the remote side) that others will see, or that could affect production processes. You'll work through an example of using the amend option in Chapter 5.

When Is the Staging Area Used by Git?

In addition to users performing actions that directly cause content to be moved into the staging area, Git also uses the staging area itself on certain occasions, notably for dealing with merge conflicts. This case most closely aligns with the *prepare* scenario I outlined previously.

Merging is significant enough functionality in Git that it gets a full treatment in Chapter 9. For my purposes here, I'll describe how it works at a high level and particularly how it uses the staging area.

When you merge in Git, you are generally merging together two or more branches. In a best-case scenario (not too uncommon), the merge may have no conflicts and everything merges cleanly. In that case, Git both completes the merge locally (in your working directory), and promotes the merged content automatically into the local repository—and you're done.

However, in a case where there are merge conflicts that Git cannot automatically resolve, Git puts those files in your working directory for you to fix, and stages any files that merged cleanly. What it is doing is starting to create a set of merged content to be committed once everything is resolved.

From here, the idea is that the user goes into the working directory and edits the files with conflicts in order to fix them. Then those fixed files are added into the staging area with the ones that were automatically merged. After this, the staging area will contain the full set of resolved files and a single commit can be done to complete the merge.

There is another side benefit of this arrangement. After the merge has been attempted, if there are conflicts, the merged files are grouped together in the staging area. Separately, the files with merge conflicts are grouped together in the working directory. This offers a very easy way to see which files fall into which category, and thus an easy way for the user to understand what is merged and what needs to be manually resolved.

Can I Bypass the Staging Area?

While the staging area is very useful for the situations outlined previously, outside of those situations, most users still want to know if they can bypass it in normal use. The answer is … usually. Git provides a shortcut method to promote files to the staging area and then to the local repository with one operation. The caveat, though, is that this only works for files that Git is already tracking, meaning that the first time a file is added to Git, it has to go through the staging process. Afterward, for normal commit operations, you can use the shortcut if you choose to simplify updating revisions. The shortcut is explained in Chapter 5.

MERGING AND THE STAGING AREA

One other area where the staging operation is required is when you need to complete a merge operation that had conflicts. As discussed in the previous section, Git stages files that merged successfully. In order to complete the merge, files that have conflicts manually resolved must be staged. This creates a complete set of content to be committed to complete the merge operation.

Other Names for the Staging Area

One other note about the staging area is that it has a couple of other names in Git. It is sometimes referred to by the terms *index* or *cache*. In fact, some Git commands will have variations of *index* or *cache* as options for operations that work on content in the staging area. For purposes of what you're doing in this book, you can think of all of these terms as meaning the same thing.

The Local Repository

The local repository is the final piece of the set of Git levels that exist on a user's local machine (the local environment). Once content has been created or updated and then staged, it is ready to be

committed into the local repository. As mentioned earlier, this repository is physically stored inside a separate (normally hidden) subdirectory normally within the root of the working directory. It is created in one of two ways: via a clone (copy) of a repository from a remote, or through telling Git to initialize a new environment locally.

The nice thing about the local repository is that it is a source repository exclusively for the use of the current user. Modifications can be done until the user is satisfied with the content, and then the content can be sent to the remote repository where it is available to others. As noted before, because everything is local, source control operations can be done to the local repository without network overhead, and even when the machine is not connected to a network.

Of course, there are always tradeoffs. Having everything local means that content is lost if the working directory is accidentally wiped out and content has not been synched to the remote repository. It also implies that the longer the time between when content is synched to the remote repository, the higher the chance of merge issues if others are continuing to update that particular remote repository.

The Remote Repository

The remote repository is the level of Git that hosts and serves up content for wider consumption. It's the place where multiple Git users sync up the changes from their respective local repositories. It corresponds to what you would traditionally think of as the server in other source management systems. I will go into more detail on remote repositories in later chapters, but there are a few general points that are useful to understand up front about remote repositories:

- A remote repository is unique. There can be many remote repositories for many different projects managed with Git, but Git does not make or use multiple copies of the remote repository on the server.
- A remote repository can be cloned as many times as needed to separate local repositories. Related to the section in Chapter 2 where I discussed the differences between centralized and distributed source management systems, multiple different users can get copies of the remote repository as their own local repositories to work with. Then, when they push changes from their local repositories, they are pushing them into the single corresponding remote repository that the local repositories were copied from.
- A remote repository does not make user-facing modifications to content, such as resolving conflicts for merging. It is primarily concerned with synching changes to and from the local repositories of individual users. If there are conflicts that need resolution at the time content is pushed over to the remote, that content has to be pulled back to the local environment, resolved there, and then synched up to the remote.

REPOSITORY BRANCHES

Before I leave the topic of repositories, it's worth saying a quick word about branches. As in most source management systems, Git supports the concept of branches (which I explore in detail in later chapters). In Git, there are branches that exist in the local repository (local branches) and branches that exist in the remote repository (remote branches). Synching of these branches occurs during some of the commands that I will talk about for working with remote repositories. At any point in time, one branch is active in the local environment, meaning that the files in the working directory tracked by Git came from that local branch.

The Core Git Commands for Moving Content

Now that you understand the different levels in the Git model, it's a good time to introduce the core Git commands for moving content between them. Some of these commands have already been mentioned in context. I'll just note them briefly here to help fill out an overall picture of the system. Chapter 5 will explain the local workflow in more detail, and later chapters will explain the workflow when working with the remote environment. I'll characterize these commands by which levels they interact with.

Working Directory to Staging Area

The *add* command stages content from the working directory to the staging area. Contrary to what the name implies, you always use the add command to stage anything, even content that is not new and that has been staged before.

Staging Area to Local Repository

The command that is used to promote things from the staging area to the local repository is the *commit* command. Think of it as making a commitment to put your changes into the official source management repository. This is most similar to what you might see as *check-in* in other source management systems, but note that it only takes content from the staging area.

Local Repository to Remote Repository

To synchronize changes from a local repository to the corresponding remote repository, the command is *push*. Unlike commits into the local repository, merge conflicts from content pushed by other users can be encountered here. Also, being able to push to a particular remote repository assumes appropriate access and permissions via whatever protocol and permissions checking is being used.

Local Repository to Working Directory

The *checkout* command is used to retrieve content (as flat files) from the local repository into the working directory. This is usually done by supplying a branch name and telling Git to get the latest copy of content from that branch. Checkout also tells Git to switch the branch that you are currently working with.

Remote Repository to Local Environment

When moving content from the remote repository to the local environment, there are several ways the local repository and the working directory can receive content from the remote repository.

The *clone* command is used to create a new local environment from an existing remote repository. Essentially, it makes a local copy of the specified remote repository onto the local disk and checks out a flat copy of the files from a branch (typically master, although this is configurable) into the working directory.

The *fetch* command is used to update the local repository from the remote repository. More specifically, it is updating reference copies of the remote branches (*reference branches*) that are maintained in the local repository. This allows for comparison between what you have in your local repository and what the remote repository had the last time you connected to it. A merge or rebase (merge with history) can then be done to update local branches as desired.

The *pull* command does a *fetch* followed by the merge or rebase (merge with history) operation. This one command then results in not only updating the reference branches in the local repository from the remote side, but also merging that content into the local branch or branches. It also updates any of the corresponding files in the working directory if the current branch is one that had updates merged from the remote side.

Table 3-1 summarizes the levels and commands.

TABLE 3-1: Core Commands for Moving Content between Levels in Git

FROM	TO	COMMAND	NOTES
Working Directory	Staging Area	Add	Stages local changes
Staging Area	Local Repository	Commit	Commits only content in staging area
Local Repository	Remote Repository	Push	Syncs content at time of push
Local Repository	Working Directory	Checkout	Switches current branch
Remote Repository	Local Environment	Clone	Creates local repository and working directory
Remote Repository	Local Repository	Fetch	Updates references for remote branches

TABLE 3-1 *(continued)*

FROM	TO	COMMAND	NOTES
Remote Repository	Local Repository and Working Directory	Pull	Fetches and merges to local branch and working directory

Putting this table into a visual representation, you can add the commands to the previous picture of the Git model. This provides a representation of Git in one picture, as shown in Figure 3-4.

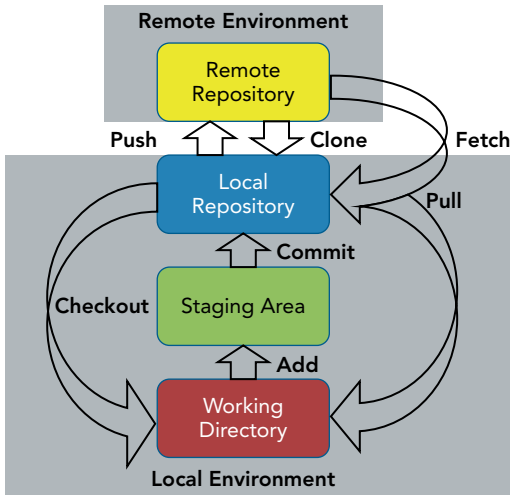


FIGURE 3-4: Git in one picture

SUMMARY

In this chapter, you looked at the Git promotion model, a way of thinking about the different levels of Git and how content is moved between them. These levels include the remote repository, the local repository, the staging area, and the working directory. The last three levels make up what I refer to as your local environment, where you develop content and do source management locally before making it more widely available by pushing to the remote.

I dove in to explain why the staging area exists, and some of the different uses and functionality it provides. These include gathering up changes for a commit (prepare), updating the last commit (repair), and providing separation between files that merge cleanly and files that don't when doing a merge.

I concluded by giving a brief summary of the commands that you use with Git to move between the different levels, leading to a single-picture representation of the Git model.

In the next chapter, you'll look at how to configure Git's various options and settings and actually start using Git to work through the model.

About Connected Lab 1: Installing Git

Before going to the next chapter though, this is a good point to work through Connected Lab 1: Installing Git. Having an installation of Git will be a prerequisite for the rest of the Connected Labs in the book. I highly encourage you to work through the labs if you are not familiar with the topics. This will provide you with hands-on experience with Git and help to internalize the concepts we discuss in the text. Enjoy!

Connected Lab 1

Installing Git

This lab will guide you through the steps for installing Git on your system. If you already have Git installed, you can skip to the next chapter. Otherwise, select the appropriate section for your operating system, and follow the instructions.

INSTALLING GIT FOR WINDOWS

The Git for Windows package installs a version of Git that also includes a Bash (Unix) shell that runs on top of Windows and provides a Unix-style interface to Git. You can also integrate Git with the Windows Explorer and command prompts.

The following instructions provide the necessary steps for installation, as well as additional information on the install screens you will encounter during the process.

STEPS

Each of the following numbered steps represents a new screen of the installation tool.

1. In your browser, go to `http://git-scm.com/download/win`. The download starts automatically.
2. After the download completes, double-click the executable file (or select the "Run" button if one is available) to start the install (If any security prompts come up, answer them to allow the install to run.)
3. Click Next after viewing the license agreement.
4. (Optional) Deselect any integration pieces you don't want. This allows you to set up integration with the Windows Explorer and file associations if you want. Click Next.
5. Select how you want to use Git. This screen gives you several options:
    a. **Use Git from Git Bash only:** This refers to the Unix shell that comes as a separate program with Git for Windows. If you are comfortable with Unix, or you aren't and don't intend to run many operating system commands, this is the simplest

option. The shell features some nice color-coding that can be helpful as you're learning Git. This option *won't* allow you to use Git integration in Windows command prompts.

b. **Use Git from the Windows Command Prompt:** The main purpose of this option is to allow you to run Git commands in Windows command prompts. It also includes the ability to use Git through the Git Bash shell. It does not try to provide full integration with some of the Unix applications in command prompts as the first option does. This is a good default because it provides the flexibility to use Git in either or both the Bash shell and Windows command prompts.

c. **Use Git and optional Unix tools from the Windows Command Prompt:** This option provides some additional Unix-style tools for you to use from command prompts. Keep in mind that these tools will be in the path and found before some of the Windows commands of the same name. In general, if you want to use Unix commands and tools, you're better off doing so through the Bash shell interface.

6. If you have access to Plink (PuTTy link) on your system, you see an additional screen allowing you to choose which *SSH executable* you want to use here. Unless you have a specific reason to do otherwise, choosing the *Use OpenSSH* option is fine.

7. Configure the line ending conversions. This refers to how you want Git to handle line endings in files when getting content out of Git or putting content into Git. I cover this setting in detail in Chapter 4. You can jump ahead and read about that now if you want, but briefly, this relates to how you plan to edit files you'll be managing with Git. You can find more details on the different options in the following paragraphs.

If you plan to use Windows editors, then the first setting—*Checkout Windows-style, commit Unix-style*—will probably work best. This setting means that when you get text content out of Git, Git updates the line endings in the checked-out files to be carriage-return/line-feed (CRLF). This is the line ending expected by Windows editors. When you check in (or *commit*) content back into Git, Git strips out the CRs and stores (normalizes) the text/ASCII files with line endings that are just LF (the default for Unix).

On the other hand, if you plan to edit with Unix-based editors (vi or others) or work primarily through the Bash shell, then the *Checkout as-is, commit Unix-style line endings* setting may be the best choice. This doesn't make any changes to the files on checkout, but normalizes them to LFs when storing them in Git. So, essentially, they will always have LFs. Because LF-only is the default for Unix systems, this works well for editing in that environment.

The last choice—*Checkout as-is, commit as-is*—can be problematic. Basically, this tells Git not to make any changes for line endings—just to leave them as they are. This means that you can end up with a mixture of line endings in the repository. The other two options normalize files in the repositories to LFs. If a file is edited in a Windows editor and then stored back in Git, the file stored in the Git repository will contain CRLFs. However, if edits are done in Unix editors, the files will have just LFs stored in the repository. If someone then gets one of these files out on an OS that is *different from* the OS where it was last edited, they may be surprised by the line endings being in the style

of the other OS. This can be especially troublesome for teams where some members use Unix and other members use Windows.

You can change this setting at a later time by changing the configuration value for *core.autocrlf* that is mentioned here. (I cover this in more detail in Chapter 4.) However, at that point, there may already be files stored in the repository with undesired line endings. There are ways to fix these files that are beyond the scope of this discussion.

For most of the work you'll do in the Connected Labs for this book, the value of this setting won't be significant. However, the best practice here is to choose one of the first two settings that best corresponds to the OS type where you plan to run your editors.

8. Configure the terminal emulator for the Bash shell. You have a choice of which terminal program you want to use for the Bash shell. Unless you have a specific reason to choose the *Windows default console window* option, choose the *Use MinTTY* option. This gives you a better user interface that supports functions such as Copy and Paste in the expected way (highlight and select) rather than with the limited functionality of the console window option.

9. Configure extra options. The *Enable file system caching* option is a relatively new addition. It attempts to speed up file-related operations for users of Git on Windows, where the file handling is not optimized in the same way as it is for Unix. In principle, this seems like a good option, although most users have had limited experience with it. Note that it can be turned off later by changing the *core.fscache* configuration value. This is one option you should probably turn on until any issues are found with it.

The *Git Credential Manager for Windows* is a successor to a previous credential management application. It essentially helps with managing and simplifying different types of access for Git from various applications. Its use is generally transparent to the user. You can read more about this application in the README file for the project on the GitHub hosting site at `https://github.com/Microsoft/Git-Credential-Manager-for-Windows/blob/master/README.md`.

Unless you have a specific reason not to use this application, just leave it checked.

10. Once you've completed the option screens in these steps, click the Install button. Git removes any old installs (if they exist) and updates with the newest version. Afterward, you have a new *Git* category in your available programs list with entries to start the Git Bash shell (the Unix shell), a Git CMD window, and a GIT GUI interface.

The Git CMD window is like a Windows command prompt. However, you can start up a Windows command prompt and also have access to Git in this window.

11. Open the Git Bash shell, the Git CMD window, or a Windows command prompt, and type

```
$ git --version
```

to make sure you have Git installed and are running at the expected version.

USING THE GIT BASH SHELL

When you start up the Git Bash shell, it generally opens up to a directory of "/" (forward slash) at the prompt. This *root directory* corresponds to the directory on your Windows file system where Git is installed. By default, that is *C:\Program Files\Git*. It's not good practice to store repositories under Program Files, so you want to switch to a different directory before starting to work with Git. In the Git Bash shell, to change to a different directory on your C drive, you use a command like this:

$ cd /C/Users/<username>

This corresponds to the following command in a regular Windows command line interface:

cd C:\Users\<username>

So, in the Bash shell, the syntax for navigating around is to represent drive letters as

/<drive letter>/ instead of *<drive letter>:* (note the slash versus colon)

and then use forward slashes instead of backward slashes in the remaining parts of the path.

Note that "~" corresponds to *c:\users\<username>* and, as previously mentioned, "/" by itself corresponds to the directory where Git was installed.

INSTALLING GIT ON MAC OS X

1. In your browser, go to `http://git-scm.com/download/mac`. The download starts automatically. If not, there is a link you can click to start it.
2. Install the downloaded file via the DMG and PKG files.
3. Open up a terminal, and run the following command to make sure Git is installed and running at the expected version:

   ```
   $ git --version
   ```

INSTALLING GIT ON LINUX

1. In your browser, go to `http://git-scm.com/download/linux`.
2. Follow the instructions on that page for the particular flavor of Linux you're using.
3. Confirm that Git is installed by opening up a terminal session and running the following command:

   ```
   $ git --version
   ```

PART II
Using Git

- **CHAPTER 4:** Configuration and Setup
- **CHAPTER 5:** Getting Productive
- **CHAPTER 6:** Tracking Changes
- **CHAPTER 7:** Working with Changes over Time and Using Tags
- **CHAPTER 8:** Working with Local Branches
- **CHAPTER 9:** Merging Content
- **CHAPTER 10:** Supporting Files in Git
- **CHAPTER 11:** Doing More with Git
- **CHAPTER 12:** Understanding Remotes—Branches and Operations
- **CHAPTER 13:** Understanding Remotes—Workflows for Changes
- **CHAPTER 14:** Working with Trees and Modules in Git
- **CHAPTER 15:** Extending Git Functionality with Git Hooks

4

Configuration and Setup

WHAT'S IN THIS CHAPTER?

- Git command syntax and format
- The differences between porcelain and plumbing commands
- Working with auto-completion
- Basic configuration of Git and your user environment
- Creating a new repository
- Dealing with line endings with Git
- The contents of a Git repository
- Creating Git aliases

When starting to use Git, it's important to configure it so that it works properly in your particular environment. You'll also want to be able to manage your content and your interactions with Git in a way that you prefer. In this chapter, you will learn how to configure your Git environment, and explore the different considerations that come into play. You'll look at some of the key required items such as line endings, as well as some of the more significant optional settings. You'll also learn how to define settings within the different scopes that Git allows.

In the "Advanced Topics" section, I'll describe how the init command works, offer more detail about what's actually in the underlying repository, and show you how to create aliases that take parameters that can run small programs.

EXECUTING COMMANDS IN GIT

As I previously mentioned, this book focuses on the Git command line to provide the most universally applicable way to use the tool. The general form of commands is as follows:

```
git <git-options>  <command>  <command-options>  <operands>
```

Table 4-1 describes the different parts of this form.

TABLE 4-1: Components of a Git Command Line Invocation

ELEMENT	DESCRIPTION	EXAMPLE(S)	NOTES
git	Command to run Git	git	
<git-options>	Global options for Git itself. These options may also specify a function to execute.	git --work-tree=<path> git --version	Some of these options may be intended for standalone operation (for example, --version), while others modify values used for other commands (for example, --work-tree).
<command>	Git command to execute	git push	
<command-options>	Options to the specified command	git commit -m "comment"	May have default options if none are specified. Options may also have values that can be selected to further qualify the option.
<operands>	Items for the command to operate on	git add *.c	Particular to the command being executed. Examples include files in the working directory, branches or SHA1s in a repository, or a particular setting or value.

Operand Types

As referenced in Table 4-1, Git can take different kinds of operands, which are specifications of objects to operate on. The two most common operands are the SHA1 value of a commit (or a named branch or tag that refers to such a commit) and a path specification to a file or directory on the disk. For many commands, either or both of these value types may be specified—or neither. When neither operand is specified, the command will operate against all eligible items that it finds in the scope of the repository, staging area, or working directory tree.

> **NOTE** *Throughout this book, I won't usually supply optional commits or path specifications to Git's commands unless they are required. This will help to simplify examples and allow you to learn about the commands in this context. However, I will introduce the forms of the commands when I first discuss them so you'll be able to see where those items can be supplied.*

The primary reason to specify both commit references and paths would be to select certain paths that are part of, or in the scope of, the snapshot associated with the commit. Because Git operates at

the granularity of a snapshot (tree), you may not always want to do the operation against all items in the snapshot. However, that's what would happen if you just specified the commit | tag | branch. To indicate that the operation should only be done against certain files or paths in the scope of the snapshot, you need to add specific filenames or paths.

When both types are specified, if there is a possibility of Git not being able to tell the difference between a commit | branch | tag and one or more of the filenames or paths, then you can separate the two types using the special separation symbol "--". Normally, this won't be needed if a commit is expressed as a SHA1 value, but it may be needed if branch or tag names could be mistaken as names for files or paths.

As an example, the command `git <command> a1b2c3d4 file1.txt` might be clear enough, but `git <command> my-tag-name -- my-file-name` could be ambiguous enough when parsed to require the "--" separator symbol.

> **NOTE** *As referenced in Table 4-1, Git has global options—in fact, quite a few. Beyond the obvious ones, such as --version and --help, there are a number of options concerned with allowing users to specify different paths for different areas of Git, as well as a few miscellaneous ones. At this point, I won't go into further detail about these options because many of them wouldn't make sense without additional context. However, where I identify value for individual options in the context of later chapters, I'll focus in on selected ones then.*

Porcelain versus Plumbing Commands

In this section, *command* represents any of the commands available in Git, such as the ones I talked about for moving content between the levels of Git in Chapter 3 (add, commit, push, and so on). In Git, there are two categories for the types of commands: *porcelain* and *plumbing*. Those names may sound strange, but essentially, the porcelain commands are intended to be user-facing, more commonly used, and more convenient. They also typically provide a higher level of functionality. The commands that I previously mentioned in conjunction with the Git promotion model are examples of porcelain commands.

The plumbing commands function at a lower level and are not expected to be used by the average user. These commands are typically targeted at extracting or modifying content and information more directly from the repository. An example would be the `git cat-file` or `git ls-files` commands that provide a way to look at the contents of a file or directory within the repository if you know how to reference those elements.

Certain functionality in Git can be accomplished using either porcelain commands or plumbing commands. However, it would usually take several very specific plumbing commands to accomplish what one porcelain command can do. The porcelain commands are based on the plumbing commands. They aggregate the functionality of plumbing commands and certain options and sequences in order to make things simpler for the typical Git user.

Table 4-2 shows a categorization of the porcelain (user-friendly) commands that are available in Git.

TABLE 4-2: Porcelain Commands in Git

COMMAND	PURPOSE
add	Add file contents to the index.
bisect	Find by binary search the change that introduced a bug.
branch	List, create, or delete branches.
checkout	Switch branches or restore working tree files.
cherry	Find commits yet to be applied to upstream (branch on the remote).
cherry-pick	Apply the changes introduced by some existing commits.
clone	Clone a repository into a new directory.
commit	Record changes to the repository.
config	Get and set repository or global options.
diff	Show changes between commits, commits and working tree, and so on.
fetch	Download objects and refs from another repository.
grep	Print lines matching a pattern.
help	Display help information.
log	Show commit logs.
merge	Join two or more development histories together.
mv	Move or rename a file, directory, or symlink.
pull	Fetch from, or integrate with, another repository or a local branch.
push	Update remote refs along with associated objects.
rebase	Forward-port local commits to the updated upstream head.
rerere	Reuse recorded resolution for merged conflicts.
reset	Reset current HEAD to the specified state.
revert	Revert some existing commits.
rm	Remove files from the working tree and from the index.
show	Show various types of objects.
status	Show the working tree status.
submodule	Initialize, update, or inspect submodules.
subtree	Merge subtrees and split repositories into subtrees.
tag	Create, list, delete, or verify a tagged object.
worktree	Manage multiple working trees.

Table 4-3 shows the same categorization for the plumbing commands. These commands have names that indicate an action and an object to operate against as opposed to the simpler naming of the porcelain commands.

TABLE 4-3: Plumbing Commands in Git

COMMAND	PURPOSE
cat-file	Provide content or type and size information for repository objects.
commit-tree	Create a new commit object.
count-objects	Count an unpacked number of objects and their disk consumption.
diff-index	Compare a tree to the working tree or index.
for-each-ref	Output information on each ref.
hash-object	Compute object ID and optionally create a blob from a file.
ls-files	Show information about files in the index and the working tree.
merge-base	Find as good common ancestors as possible for a merge.
read-tree	Read tree information into the index.
rev-list	List commit objects in reverse chronological order.
rev-parse	Pick out and massage parameters.
show-ref	List references in a local repository.
symbolic-ref	Read, modify, and delete symbolic refs.
update-index	Register file contents in the working tree to the index.
update-ref	Update the object name stored in a ref safely.
verify-pack	Validate packed Git archive files.
write-tree	Create a tree object from the current index.

The descriptions for the commands in these tables are taken directly from the Git help. Some of the terms are more Git-specific at this point. However, as I use commands through the remainder of this book, I'll simplify their definitions and the terminology so it all makes sense.

The point of this section is that unless you have a specific need to deep-dive into the repository, you can simply use the porcelain commands and accomplish what you need to in Git.

Specifying Arguments

Arguments supplied either to Git or to Git commands can be abbreviated as a single letter or spelled out as words. One important note here is that if the argument is spelled out, you must precede it with two hyphens, as in `--global`. If the argument is abbreviated, only one hyphen is required, as

in *-a*. Abbreviated arguments may be passed together, as in `-am` instead of `-a -m`. When arguments are combined in this way, the ordering is important. If the first argument requires a value, then the second argument may be taken as the required value instead of an additional argument.

Auto-complete

When you start typing a command or an argument to a command, Git has a helpful auto-completion feature (if enabled) that can do two things:

- Provide valid values for the commands or arguments that could complete the text you're typing—if there is more than one valid option.
- Automatically complete the command or argument that you're typing—if there is only one valid option.

Following are a couple of examples. The first one is for a command. If you type `git c` and then press the Tab key, nothing happens because there's more than one command that starts with *c*.

If you press the Tab key a second time (before typing anything else in between), Git helpfully displays all of the commands that start with *c*. In this case, it also scrolls that list up and leaves you at a prompt where you can continue typing the chosen command.

```
$ git c
checkout      citool        commit
cherry        clean         config
cherry-pick   clone

$ git c
```

Here's another example, where you narrow the available commands with more letters.

```
$ git co <TAB><TAB>
commit   config

$ git c
```

If you type enough letters to uniquely identify only one possible choice, then pressing the Tab key auto-completes the command for you because there's only one option. For example, `git con <TAB>` yields `git config`.

This also works for arguments to commands. Typing `git config --l <TAB> <TAB>` gives the suggestions: `--list  --local`. Typing either `git config --l` or `git config --li <TAB>` yields `git config --list`.

> **NOTE** *When attempting to use auto-complete for an option, make sure that you have started the option with the double-hyphen (--) syntax and not just a single hyphen.*

Enabling Auto-complete If You Don't Have It

As noted earlier, auto-complete is already enabled in Git for Windows and some other distributions. For other versions (Linux, OS X) where it is not enabled, you can download scripts that implement

this feature for different shells from `https://github.com/git/git/tree/master/contrib/completion`.

Once you understand tools like git pull, you can use them to retrieve these scripts via Git. Until then, or as an alternate approach, a simple way is just to click the desired script and then find the button labeled *Raw* on that page. Click that button to go to a web page with just the contents of that file. Then, you can download that script to your local system (through the browser) and add it into the appropriate init file in your home directory or into the appropriate directory for auto-completion for all users if your shell supports that.

Let's work through a quick example of how to install this feature for a bash environment.

Here's the direct link for the raw version: `https://raw.githubusercontent.com/git/git/master/contrib/completion/git-completion.bash`

After getting the raw version of the file, you can download that page as the file `git-completion.bash` to your local system. Once the script is downloaded, you add a line like the following into your .bashrc file (create the file if needed):

```
$ source ~/git-completion.bash
```

To extend this functionality for all users, you'll need to find out where your particular OS stores and expects to find auto-completion scripts and put the downloaded file there. For most bash systems, there is a `/etc/bash_completion.d` directory where scripts like this can be stored to be loaded. If you're not sure where the location is, try searching for *completion* on your file system, or consult Google.

Auto-completion and the Windows Command Prompt

In the Windows command prompt, auto-complete functionality is not built in, and the method in the previous section doesn't work because it is based on a Linux script. However, there is a utility called *clink* that you can search for, download, and install on Windows that will provide command auto-completion for Git (as well as other functionality). The use is the same—suggestions or completion via the tab key.

Note, however, that this does not provide suggestions or auto-completion for arguments to the commands.

Now that you understand how to invoke Git commands and pass arguments, let's see how you can use this feature to accomplish one of the most basic and essential parts of using Git: configuration.

CONFIGURING GIT

To set configuration values in Git, you use the `config` command. Here's the syntax:

```
git config [<file-option>] [type] [--show-origin] [-z|--null] name [value [value_
regex]]
git config [<file-option>] [type] --add name value
git config [<file-option>] [type] --replace-all name value [value_regex]
git config [<file-option>] [type] [--show-origin] [-z|--null] --get name [value_
regex]
git config [<file-option>] [type] [--show-origin] [-z|--null] --get-all name
[value_regex]
```

```
git config [<file-option>] [type] [--show-origin] [-z|--null] [--name-only] --get-
regexp name_regex [value_regex]
git config [<file-option>] [type] [-z|--null] --get-urlmatch name URL
git config [<file-option>] --unset name [value_regex]
git config [<file-option>] --unset-all name [value_regex]
git config [<file-option>] --rename-section old_name new_name
git config [<file-option>] --remove-section name
git config [<file-option>] [--show-origin] [-z|--null] [--name-only] -l | --list
git config [<file-option>] --get-color name [default]
git config [<file-option>] --get-colorbool name [stdout-is-tty]
git config [<file-option>] -e | --edit
```

Now here's an example of the most common syntax:

```
$ git config --global user.name "Joe Gituser"
```

Let's dissect the various parts of this command. The first two pieces are simply issuing the *config* command from git. After that is an option, *global*, (preceded by two hyphens because you are spelling it out). I'll be talking in more detail about this option shortly. Next comes the configuration setting that you're updating: *user.name*. Git uses a "." notation to separate out the two pieces of a configuration setting—in this case, *user* and *name*. Think of this as setting the name *value* of the user *section* in the configuration. And finally, you have the actual value that you're setting this configuration setting to. Notice that because you have spaces in the value, you need to enclose the entire string in quotes.

Here's another example:

```
$ git config --global user.email Joe.Gituser@mailhost.com
```

One additional note: Git configuration settings are stored in text files. It is possible to change these settings by editing the associated text files, but this is highly discouraged because it's easy to make a mistake and also to accidentally modify other settings.

Telling Git Who You Are

Referring to the two earlier examples, one of the first things that you need to configure in Git is who you are, in terms of the username and e-mail address. Git expects you to set these two values, regardless of what interface or version of Git you use. This is because Git is a source management system. Because its purpose is to track changes by users over time, it wants to know who is making those changes so that it can record them.

If you don't specify these values, then Git will *interpolate* them from the signed-on userid and machine name (user@system). Chances are this is not what you want to have the system ultimately use. If you forget to set these values initially on a new system, and commits are recorded with the interpolated values, there is a way to go back and correct this information, using the commit command with the --amend and --reset-author options.

The values can be set via the same commands as shown in the previous section: `git config --global user.name <name>` and `git config --global user.email <email address>`.

> **NOTE** *The e-mail address is not validated when you set it in Git. In fact, you can enter any e-mail address and Git will be happy. However, there is some advanced functionality in Git that uses this e-mail address. That functionality allows for tasks such as creating and sharing patches and zipped versions of changes. For that functionality, having a correct e-mail address is important. Also, there are other tools, such as Gerrit (a code-review tool built on top of Git), that heavily utilize the e-mail address and depend on it being correct.*

Configuration Scope

In the previous examples, I used the --global option as part of the configuration step. The global option is a way of telling Git how broadly this configuration setting should be used—which repositories it should apply to.

Recall that the Git model is designed for many, smaller repositories instead of fewer, monolithic ones. Because users may normally be working with multiple repositories, it would be inconvenient and subject to error to have to configure the same settings in each repository. As a result, Git provides options to simplify choosing the scope for configuration values. There are three levels available for configuration: system, global, and local.

System

Configuration at the system level means that a configuration value applies to all repositories on a given system (machine) unless it's overridden at a lower level. These settings apply regardless of the particular user.

To ensure that a configuration value applies at the system level, you specify the `--system` option for the config command, as in `git config --system core.autocrlf true`.

These settings are usually stored in a gitconfig file in either `/usr/etc` or `/usr/local/etc`. On a Windows system, if you're using Git for Windows, the system file is in `C:\ProgramData\Git\config`. In other systems, look in the directory where Git was installed.

Global

Configuration at the global level implies that a configuration value applies to all of the repositories for a particular user, unless overridden at the local level. Unless you need repository-specific settings, this is the most common level for users to work with because it saves the effort of having to set values for each repository. An example of setting values at the global level would be the configuration I did earlier for `user.name` and `user.email` where the `--global` option was incorporated. These settings are stored in a file named `.gitconfig` in each user's home directory.

Local

Setting a configuration value at the local level means that the setting only applies in the context of that one repository. This can be useful in cases where you need to specify unique settings that are particular to one repository. It can also be useful if you need to temporarily override a higher-level setting.

An example could be overriding the global end of line settings because content in a repository is targeted for a different platform. To update settings at this level, you can specify the --local option or just omit any of the local, global, or system options for the configuration.

As an example of this last point, the following two commands are equivalent: `git config --local core.autocrlf true` and `git config core.autocrlf true`.

The local repository's configuration is stored within the local Git repository, in `.git/config` (or in `config` under wherever your Git directory is configured to be.)

These *scope options* (--local, --global, and --system) can be applied to other options and forms of the git config command to indicate the scope to be referenced for that command.

Settings Hierarchy

When determining what configuration setting to use, Git uses a particular *search order* to find these settings. First, it looks for a setting in the local repository configuration, then in the global configuration, and finally in the system configuration. If a specific value is found in that search order, then that value is used. Beyond that, the union of all of the levels (unique local + unique global + unique system) forms the superset of configuration values used when working with a repository.

Figure 4-1 summarizes the different configuration scopes in Git and how to work with them.

Seeing Configuration Values

To see what value a particular configuration setting has, you can use `git config <setting>` as in `git config user.name`.

Git then prints the value associated with that setting. Because I didn't specify one of the scope options (--system, --global, --local), Git first checks to see if there is a local setting, and if so, it displays that value. If there is no explicit local setting, then it looks for a global setting, and, if one is found, displays the global value. If there is no global setting specified, Git looks for a system setting and displays that value. This is an example of the search order that I outlined earlier.

You can also use the scope options to specifically direct the config command to a particular level, as I did when setting configuration values earlier.

To better understand how this works at a practical level, consider the following sequence:

```
$ git config --global user.name "Global user"
$ git config user.name
```

This returns the value *Global user* because there was no local value defined; Git looked for a global setting and found this one.

On the other hand, say you were to use this sequence:

```
$ git config user.name "Local user"
$ git config user.name
```

This returns the value *Local User* because the local option was implied in setting the value and thus it finds a local value defined.

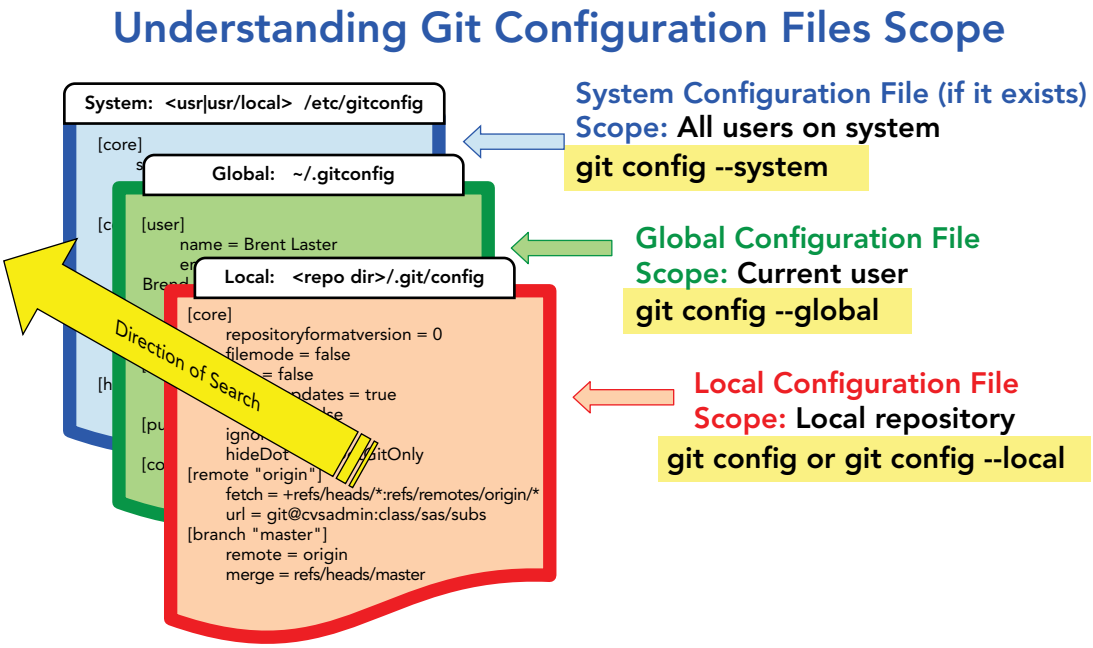


FIGURE 4-1: Understanding the scopes of Git configuration files

Undoing a Configuration Setting

Occasionally, you may need to remove a user setting at a particular level. Git provides the *unset* option for this, and it's pretty straightforward:

```
$ git config --unset <other options> <value to remove>
```

Other options here would generally refer to one of the scope options. Continuing the earlier example,

```
$ git config --unset --global user.name
$ git config --global user.name
```

In this case, nothing is returned because I just removed this value.

Listing Configuration Settings

Another option related to viewing configuration values is --list. Supplying the list option to git config results in a list of all configuration settings being dumped. By default, this list includes local, global,

and system settings without qualification. So, if you have both a local and global value for the same setting, you will see both.

```
$ git config --list
...
user.name = global user
...
user.name = local user
```

If the settings have the same values, this can be confusing (and potentially misleading) if you're not aware of the reasons behind it. To work around seeing these multiple values, you can refine the list by specifying one of the scope options.

```
$ git config --local --list
...
user.name = local user
```

> **NOTE** *If you are ever unable to figure out where a particular configuration value is set, you can use the* `--show-origin` *option with the configuration setting name to figure it out. For example, if you run the command* `git config user.name "Joe Gituser"` *then* `git config --show-origin user.name` *shows this:* `file:.git/config  Joe Gituser`.
>
> *This option can also be combined with the* --list *option to get a complete list of where all the settings are stored.*

One-Shot Configuration

There is one additional way to set a configuration value: as a one-shot, one-time configuration for the current operation. This is done through one of the global options that can be passed to Git directly: -c.

The format for this is `git -c <configuration setting>=<value> <rest of command line>`.

Notice that this format requires the "=" sign between the setting and the value. Using this option effectively creates an override for the duration of the current operation.

Now that you understand how configuration settings are specified and managed in Git, let's look at configuration for some of the most common settings and behaviors that users deal with.

> **NOTE** *To see a list of the different settings and values that can be configured, see the man page for git-config under the "Variables" section.*

Default Editor

The default editor is primarily used when you need to type in a message while making a commit into the repository. If you don't supply the message in the command line when you do the commit, Git will bring up the default editor on your system to allow you to type one in.

If you would rather use a different editor, you can use the following config command to specify which one to use: `git config --global core.editor <editor name or path + name> <optional options for the editor>`.

The `--global` option is not required, but most users want to use the same editor for all of their repositories. Here again, you can break down `core.editor` as the editor value in the core section of the configuration.

If the editor is already in the path that Git knows about, then the path isn't required. Here are some examples of configuring editors:

```
$ git config core.editor vim  (Linux)

$ git config --global core.editor "nano"  (OS X)

c:\> git config core.editor "'C:\Program Files\windows nt\accessories\wordpad.exe'"
(Windows)

$ git config --global core.editor "'C:/Program Files
(x86)/Notepad++/notepad++.exe' -multiInst -noSession
-notabbar"  (Bash shell on Git for Windows)
```

Note the different uses for single quotes and double quotes in the respective examples. Also, in the last example, -multInst, -noSession, and -notabbar are all options to Notepad++ to make it simpler to use. (multInst tells Notepad++ to allow multiple instances to run; noSession tells it not to remember the session state—that is, not to load the last file you were working on; and notabbar just avoids displaying the tabbed selection bar at the top.)

> **NOTE** *If you are working on Windows and want to set up the default editor automatically, you can use a utility program called GitPad. You can download it from* `https://github.com/downloads/github/GitPad/Gitpad.zip`.
>
> *Once you run GitPad, it will set Git's default editor to whatever application is set to open files of type* `txt` *on Windows. By default, that is Notepad, but it can be changed on Windows (through the file associations) so that it is a different application.*

End of Line Settings

Now, let's look at one of the key settings users need to manage with Git: handling end of line (EOL) values. Git manages the two types of line endings: carriage returns/line feeds (CRLF) for Windows and line feeds (LF) for OS X/Linux.

In the context of Git, there are two options that are controlled by the EOL setting:

- How line endings are stored in content when it is committed into the repository
- How line endings are updated (or not) when content is checked out of the repository onto a local disk

The first item refers to whether or not Git *normalizes* line endings in the repository. Normalizing refers to stripping out CRs and only storing files with LFs.

For the second item, when content is checked out of Git, Git can update line endings in text files. This option allows you to specify whether or not Git updates line endings in files after checkout, and, if it does, which type it sets them to.

At a user or repository level, how Git handles these options is controlled by a configuration setting named `core.autocrlf`. As before, the "." is a separator, and you can think of the first part as the section of the configuration, and the second part as the specific value being set in that section. The *crlf* part here obviously stands for carriage return, line-feed—meaning the common EOL sequence for files on a Windows environment. The *auto* part refers to automatically inserting CRLF sequences in files when they are checked out.

There are three possible values for the core.autocrlf setting:

> **core.autocrlf=true.** This value tells Git to normalize line endings to just LFs when storing files in the repository and to automatically insert CRLFs when files are checked out. If users are working on a Windows environment, this is the recommended value. It allows them to get CRLFs in files when checked out from Git, but doesn't store the CRs in the repository.
>
> **core.autocrlf=input.** This value tells Git to normalize line endings to just LFs when storing files in the repository but not to change anything when files are checked out. If users are working in a Unix environment, this is the recommended value because Unix expects just LFs.
>
> **core.autocrlf=false.** This default value tells Git not to change anything when files are being checked in or checked out. This is the primary value for the setting that can get users into trouble. Suppose you have two users working on code for the same repository, one in a Windows environment and one in a Unix environment. If both users have specified the `core.autocrlf=false` value in their configurations, then when they commit changes, the files from Windows will have CRLFs and those from Unix will have just LFs. If the respective users later each check out the other's files, then the files will have the wrong line endings for their system. For this reason, this value should not be used when mixed environments are being used in a project.

In general, it's a best practice to set the core.autocrlf setting to one of the values other than false, depending on which environment you're working in.

It should also be noted that there are other configuration settings that can contribute to how line endings are handled. However, these settings are more obscure and broader in terms of what they affect. Also, their default values generally work well for what most users need to do.

> **NOTE** *You cannot guarantee that everyone will have the appropriate core .autocrlf value set. However, there is an alternative method for controlling line endings in a repository: the .gitattributes file.*
>
> *I will discuss this file in more detail in Chapter 10, but essentially, this is a* metafile *that tells Git how to handle certain operations and characteristics based on the file's type. One of these characteristics is line endings.*

> *The advantage of controlling line endings in a .gitattributes file rather than relying on the configuration settings is that the file can be checked in to the repository along with the files it handles.*
>
> *Additionally, this file can also be used to tell Git which file types are binary.*

Aliases

Configuration in Git also supports the concept of configuring aliases for command strings. The format for defining an alias is `git config <scope option> alias.<name> <command string>`.

In this context:

- **<scope option>** can be one of `--system`, `--global`, or `--local`. (Or it can be omitted, to default to `local`.)
- **<name>** is the name you want to use for the alias. Once set, this can be used just like any other Git command.
- **<command string>** is the string of a command and any arguments that the alias will substitute for.

There are two main reasons that aliases are convenient to create and use:

- To save typing frequently used strings of commands and arguments
- To create a more familiar command for a Git command

As an example of the first case, the git log command displays history in Git and has many options. Here's an example log command:

```
$ git log --pretty=format:"%h %ad | %s%d [%an]" --graph --date=short
```

Because this is a long command string, it can be difficult to type each time you want to use it. So, you can create an alias instead.

```
$ git config --global alias.hist git log --pretty=format:"%h %ad | %s%d [%an]"
--graph --date=short
```

With this alias in place, you can now just type `git hist` instead of the longer, complicated command.

As an example of the second use case, suppose a user is more accustomed to typing **checkin** from using other SCMs instead of *commit*. If they want, they can create an alias by using `git config --global alias.checkin commit`.

After this, the user can type `git checkin` instead of `git commit`. (Note that, while this sort of alias can be created, it is not recommended because it is not universal and obscures Git's native commands.)

The format of the command to create an alias is consistent with the other config syntax. When you see `alias.<name>`, you can think of this as creating a value in the *alias* section of the configuration file. The alias information is stored in the configuration file for the specified scope.

Windows Filesystem Cache

The underlying filesystem layer on Windows is fundamentally different from the filesystem layer on Linux. Git's filesystem access is optimized for the Linux filesystem, and in the past, some operations on Git for Windows were noticeably slower. To compensate, Git has added a filesystem cache that does bulk reads of file system data and stores it in memory. This speeds up many operations, once the cache is initially populated. In recent versions of the install for Git for Windows, this option is turned on by default. To set it manually, you change the `core.fscache` value to `true` via `git config --global core.fscache true`.

INITIALIZING A REPOSITORY

Now that you understand how to configure the Git environment, I'll move on to setting up a local environment. Recall that a local environment consists of the three levels I discussed in the previous chapter: working directory, staging area, and local repository.

There are two ways to get a local environment for use with Git:

- Creating a new environment from a set of existing files in a local directory, via the `git init` command
- Seeding a local environment by copying an existing remote repository, via the `git clone` command

I'll discuss each of these methods in turn.

Git Init

The git init command is used for creating a new, empty Git repository in the local directory. The syntax for the command is shown below.

```
git init [-q | --quiet] [--bare] [--template=<template_directory>]
    [--separate-git-dir <git dir>]
    [--shared[=<permissions>]] [directory]
```

When this command is run, a new subdirectory named *.git* is created in the directory where the command was run, and populated with a *skeleton* repository. (Like many open-source applications, Git stores metadata in a subdirectory named for the tool and preceded by a dot.) This local environment is now ready for tracking and storing new content. Note that this command can be run at any time in a directory that does not already have a Git environment associated with it to create one, no matter how many or what types of files are already in the directory.

The basic syntax for invoking init is `git init`. Before running git init, you should be at the top level of the tree you want to put under Git control. You also want to make sure this is done at an appropriate level of granularity. Recall that Git is intended to work with multiple, smaller repositories, not very large ones. So, running git init at your home directory level, for example, is not usually a good idea because this sets Git up to try and act on all files and subdirectories under your home directory for future operations—which is probably beyond the scope you intended.

Git Clone

Whereas the init command is used when you want to create a new, empty repository and begin adding content, the clone command is used to populate a local repository from an existing remote repository. The syntax for the command is shown below.

```
git clone [--template=<template_directory>]
       [-l] [-s] [--no-hardlinks] [-q] [-n] [--bare] [--mirror]
       [-o <name>] [-b <name>] [-u <upload-pack>] [--reference <repository>]
       [--dissociate] [--separate-git-dir <git dir>]
       [--depth <depth>] [--[no-]single-branch]
       [--recursive | --recurse-submodules] [--] <repository>
       [<directory>]
```

To use the clone command, you specify a remote repository location to clone from and Git does the following:

- Creates a local directory with the same name as the last component of the remote repository's path
- Within that directory, creates a .git subdirectory and copies the appropriate parts of the remote repository down to that .git directory
- Checks out the most recent version of a branch (usually the default *master* branch) into the local directory. This checked-out version with the flat files is what the user usually sees and works with immediately after the clone.

The basic syntax for cloning a repository is `git clone <url>` where <url> is a path to a remote repository. Here's an example:

```
$ git clone ssh://admin@gitserver.domain.com:path-to-repo.git
```

I will discuss this command more in Chapter 12.

What's in the Repository

Whether the local environment is created by a `git init` or `git clone` command, the structure within the .git subdirectory is the same. Essentially, a Git repository is a content-addressable data store. That is, you put something into the repository, Git calculates a hash value (SHA1) for the content, and you can then use that value later to get something out. Figure 4-2 shows an outline of a .git repository on disk.

```
$ tree.sh -* .git
COMMIT_EDITMSG
HEAD
config
description
hooks
    applypatch-msg.sample
    commit-msg.sample
    post-commit.sample
    post-receive.sample
    post-update.sample
    pre-applypatch.sample
    pre-commit.sample
    pre-rebase.sample
    prepare-commit-msg.sample
    update.sample
index
info
    exclude
logs
    HEAD
    refs
        heads
            master
objects
    3a
        0e351dc84e32abec5d4dd223c5abdabd57b7f5
    4c
        7637a4b66aefc1ee877eaa1afc70610f0ee7cc
    80
        7805ea7ae9fdf1b06e876af6e9a69a349b52a3
    88
        bee8f0c181784d605eabe35fb04a5a443ae6b7
    ba
        2906d0666cf726c7eaadd2cd3db615dedfdf3a
    info
        *
    pack
        *
refs
    heads
        master
    tags
        *
```

FIGURE 4-2: Tree listing of a .git directory (local repository)

The HEAD file keeps track of which branch the user is currently working with. The description file is used by GitWeb, a browser application to view Git repositories. The config file is the local repository's configuration file. The object and pack directories are the areas where content is actually stored. You can find more information about the files and content stored in the local repository in the optional steps of Connected Lab 2.

ADVANCED TOPICS

In this section, I'll look at several topics. The first is a quick note about how the init command works. Second is a further explanation about what's in a Git repository. The third is how Git config statements map to the text of the configuration files. Finally, I'll look at a way to create even more useful aliases that can have arguments passed to them and do multiple steps.

While this information is not necessary for using Git, sometimes it's helpful to understand how Git works behind the scenes. The first two sections apply this approach to a couple of areas.

Git Init Demystified

If you're wondering how `git init` gets the initial content for the skeleton repository, the answer is that there's a template area containing the repository skeleton. This is installed when you install Git. If you're interested in looking at it, you can search for git-core on your filesystem in the area where you installed Git. On Windows, this is usually in a location such as `C:\Program Files\Git\mingw64\share\git-core\templates` (if you installed the Git for Windows package). On a Linux system, it may be in a location such as `/usr/share/git-core/templates`.

On some installations, you may also see a `contrib` folder in the same area with items such as hooks that users have contributed over time that are now included as optional pieces that can be put in place as desired. I'll talk more about setting up hooks in Chapter 15.

Running Git Init Twice on the Same Repository

Running init twice may seem counterintuitive, but there are actually cases where it provides value. The good news is that it does not delete or modify any content that you have added or committed into the repository or your local configuration. It does update any changes to the subset of the templated files discussed previously.

So what would be a use case to deliberately run init twice? Suppose you have multiple Git repositories on your system and you want to update a hook in all of them to provide some functionality, such as sending e-mails after a commit. You could update the hook in the templates area discussed earlier, and then do a *git init* on each of the repositories to get the updated hook put in place in each repository.

Looking Further into a Git Repository

As I've previously mentioned, a local Git repository is housed in the .git directory of the working directory. It is essentially a content-addressable database, meaning you supply a value (typically a SHA1) and you get content back out.

Figure 4-3 shows the relationship and transformation of content from the working directory into the Git repository.

Starting at the left side of the figure, files and directories first exist as normal OS items on the disk in the working directory. Git does not know anything about them. It does not track them until the user adds them to the staging area. Once they are tracked by Git, a new snapshot is created with metadata in the form of a commit record. Once committed, the pieces are stored in their respective areas in the underlying repository.

As shown in the middle section of the figure, the pieces that Git stores are defined as one of three types: blob, tree, or commit. Blobs are essentially anonymous containers for files—anonymous in the sense that they don't contain filenames. Trees can be thought of as containers for directories that point to blobs for files and contain the filenames. Commits can be thought of as the header records with meta-information that Git uses for tracking.

Internally, Git computes SHA1 checksums for each of these pieces and stores them referenced by those checksums. The checksums can be seen in the parts of the middle section and then in the tree view of the actual repository directory on the right side of the figure.

As shown in that view, Git stores these internal objects in directories that start with the first two characters of the checksum. The filename is made up of the remaining characters. The files may be changed over time when certain events trigger Git to do further compression and rearrange content to efficiently store very similar versions.

The only checksum those commands are concerned with (and the only one you need to be concerned with) is the checksum that is specifically associated with the commit record, not the ones for trees or blobs.

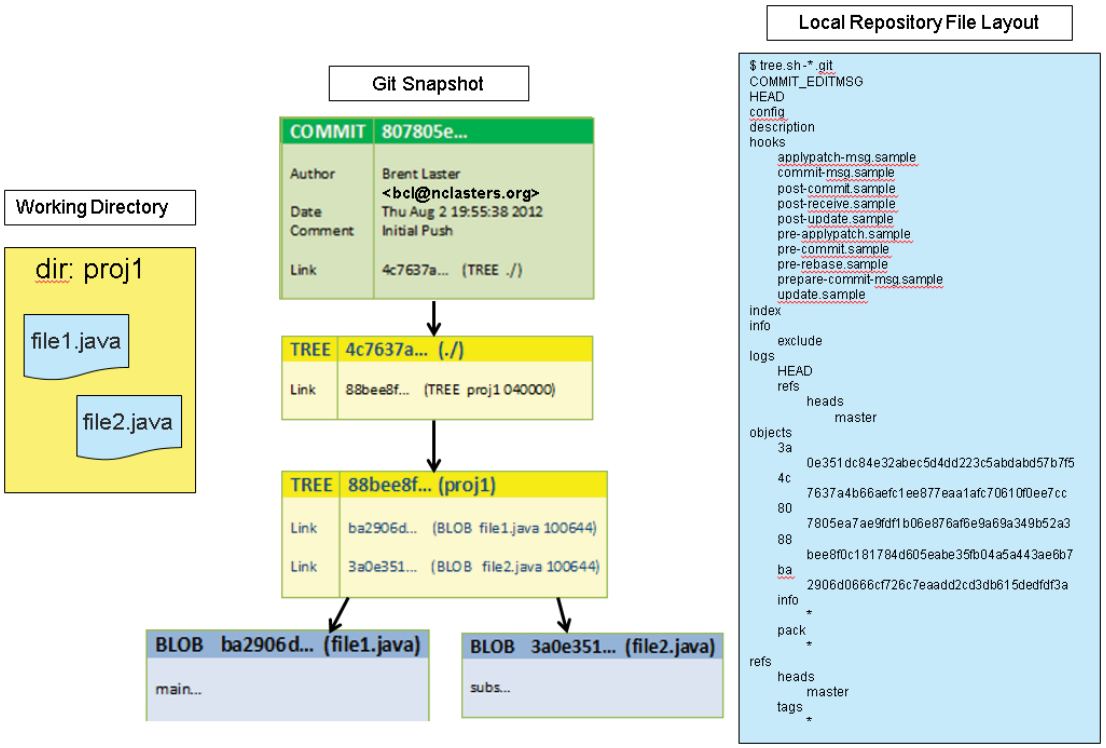


FIGURE 4-3: Mapping files and directories to Git repositories

By referencing that one checksum for the commit record, Git pulls in the underlying tree and blob content.

Once you actually have a repository with content stored in it, you can change into the repository directories to see the stored objects or use this shortcut (on Linux systems): `find .git/objects -type f`.

From there, you can use the `cat-file` plumbing command to examine objects. As an example `git cat-file -p <sha1 or branch name or reference>` tells Git to figure out the type of object

and neatly display its contents. A similar command, `git cat-file -t <sha1 or branch name or reference>` returns the type of an object: *commit, tree,* or *blob.*

Connected Lab 2 contains several optional steps that you can work through to understand what's happening in the underlying repository during an init, add, and commit sequence. It also further explains the files that are in the underlying repository tree at various points.

Mapping Config Commands to Configuration Files

In this chapter, I described the various configuration files that Git uses, as well as how to set values via the git config command. If you see something in a config file that you want to emulate or use, it can be helpful to understand how the config commands map to the file structure. This section will explain that.

Suppose you configure a two-part value such as user.name in your local configuration with a command like `git conig --local user.name "Git User"`.

This translates into setting the *name* value of the *user* section, written into the .git/config file as follows:

```
[user]
        name = Git User
```

If you need to configure a given value for a named section, you can use a three-part value such as the following:

```
$ git config --local remote.myremote.url http://github.com/brentlaster/calc2

$ cat .git/config
...
[remote "myremote"]
             url = http://github.com/brentlaster/calc2
```

Anything beyond three parts is still treated as three parts, with the extra pieces at the front just made part of the named section.

```
$ git config --local remote.myremote.new.url http://github.com/brentlaster/calc2

$ cat .git/config

...
[remote "myremote.new"]
               url = http://github.com/brentlaster/calc2
```

Note that the git config operation also takes a --file option instead of --local, --global, or --system. This allows for writing configuration options to a file in a different location, such as for test purposes.

```
$ git config --file test.config remote.myremote.test
        http://github.com/brentlaster/calc2git

$ cat test.config
[remote "myremote"]
                  test = http://github.com/brentlaster/calc2git
```

As one last tip, git config includes a --get-regexp option to find configuration values matching a specified pattern. I'll use this option in the next section so you can see how it works.

Creating Parameterized Aliases

Earlier in this chapter, I showed how to create simple aliases for specific Git command lines, such as `git config alias.ci commit`. It is certainly useful to alias fixed command strings, but only to the extent that arguments and options included in the alias never change.

What if you want to create an alias that takes a parameter that is not normally part of a command? Or that may change over time? Or that may perform extra steps or processing—especially with system commands?

As it turns out, on Linux systems you can do this with Git fairly easily. You just need to have your alias string take this form:

```
"! f() { do some processing }; f"
```

The ! at the beginning tells Git you are going to the shell. The "f() {}; f" allows you to define a function as part of the alias and then run that function when the alias is invoked.

Values that you pass in as arguments are treated as positional parameters (for example, $1, $2, and so on). When including these parameters in the alias definition, a backslash needs to precede the $, as in "\$". This is to ensure the parameter is included as part of the definition and not interpreted when you are defining the alias.

Let's work through a couple of examples. First, I'll create a simple alias that takes an argument and lists out any matching global and local settings prefixed by an appropriate header for each section. The config command is used in this example. What I am doing in this command line is defining a local alias named *scopelist*, which does the following:

1. Echoes out a *global settings* header
2. Uses git config's --get-regexp option with a global qualifier to search for the value that is passed in
3. Echoes out a *local settings* header
4. Uses git config's --get-regexp option with a local qualifier to search for the values that are passed in

Here's the command. (Pay attention to the quotes, semicolons, double hyphens, and backslashes.)

```
$ git config --local alias.scopelist "! f() { echo 'global settings'; git config
--global --get-regexp \$1; echo 'local settings'; git config --get-regexp \$1; }; f"
```

Here is an example of running the alias:

```
$ git scopelist name
global settings
user.name Git User (global)
local settings
user.name Git User (local)
```

The following example will show you a simple way to dump out the contents of a particular scope into a file. This illustrates having two positional parameters. In this case, the alias will do the following:

1. Echo out a header.
2. Issue a git config command at the appropriate scope.
3. Dump the values from step 2 into a separate file.

Here's the command to define this alias. (Again, pay attention to the punctuation characters that are used.)

```
$ git config --local alias.dumpvalues "! f() { echo 'copying config' \$1; git
config --list --\$1 > \$2; }; f"
```

Here is an example of running this alias and looking at the results:

```
$ git dumpvalues global global_values.out
copying config global

$ cat global_values.out
alias.hist=log --pretty=format:"%h %ad | %s%d [%an]" --
graph --date=short
push.default=simple
core.autocrlf=false
core.editor='C:/Program Files
(x86)/Notepad++/notepad++.exe' -multiInst -noSession -notabbar
gitreview.remote=origin
user.name=Git User (global)
user.email=Git.User@domain.com
```

Obviously, these examples don't cover all the possibilities of bad or missing input. However, they'll give you an idea of how to use this functionality if you ever need it.

SUMMARY

In this chapter, I discussed the form and structure of Git commands and related topics such as auto-completion. I introduced basic configuration for Git and described how to create local environments. I covered the different scope of configuration settings you can use and how to specify values for each scope. I also covered how to create aliases to simplify interacting with Git. I then described the two different ways to create local environments with Git—initializing a new environment from existing files or cloning down an existing repository. Finally, I offered a brief description of what's inside a .git repository.

In the section on advanced topics, you took a closer look at how the init command works, the contents of a Git repository, and how to look at individual objects. Then you learned how configuration commands map to the actual configuration text files. Finally, you saw how to create advanced aliases that can run operating system commands and allow you to work with positional parameters.

In the next chapter, you'll start putting content into Git and go over the commands to start promoting it up through the levels.

5

Getting Productive

WHAT'S IN THIS CHAPTER?

- Getting and working with help in Git
- Understanding the multiple repositories model
- Staging files
- Partial and interactive staging
- Committing files into the local repository
- Writing good commit messages

Now that you understand the Git workflow, how to create a repository, and how to configure the local environment, I'll show you how to use Git to start tracking and managing content. I'll also further explain concepts such as SHA1, options for staging files, and forming good commit messages. First, though, I'll discuss something that both new and experienced users need to know: how to get help.

GETTING HELP

Git includes two different forms of help: an abbreviated version and a full version. The abbreviated version is a short list of options with brief explanations that display one per line on the terminal screen. It is invoked by using the -h option after the command, as in `git commit -h`.

This is useful when you just need a quick reminder of what options are available or how to specify a particular option. Figure 5-1 shows an example of abbreviated on-screen help.

```
MINGW64:/
$ git commit -h
usage: git commit [<options>] [--] <pathspec>...

    -q, --quiet           suppress summary after successful commit
    -v, --verbose         show diff in commit message template

Commit message options
    -F, --file <file>     read message from file
    --author <author>     override author for commit
    --date <date>         override date for commit
    -m, --message <message>
                          commit message
    -c, --reedit-message <commit>
                          reuse and edit message from specified commit
    -C, --reuse-message <commit>
                          reuse message from specified commit
    --fixup <commit>      use autosquash formatted message to fixup specified co
mmit
    --squash <commit>     use autosquash formatted message to squash specified c
ommit
    --reset-author        the commit is authored by me now (used with -C/-c/--am
end)
    -s, --signoff         add Signed-off-by:
    -t, --template <file>
                          use specified template file
    -e, --edit            force edit of commit
    --cleanup <default>   how to strip spaces and #comments from message
    --status              include status in commit message template
    -S, --gpg-sign[=<key-id>]
                          GPG sign commit

Commit contents options
    -a, --all             commit all changed files
    -i, --include         add specified files to index for commit
    --interactive         interactively add files
    -p, --patch           interactively add changes
    -o, --only            commit only specified files
    -n, --no-verify       bypass pre-commit hook
    --dry-run             show what would be committed
    --short               show status concisely
    --branch              show branch information
    --porcelain           machine-readable output
    --long                show status in long format (default)
    -z, --null            terminate entries with NUL
    --amend               amend previous commit
    --no-post-rewrite     bypass post-rewrite hook
    -u, --untracked-files[=<mode>]
                          show untracked files, optional modes: all, normal, no.
 (Default: all)
```

FIGURE 5-1: Abbreviated version of help invoked with the -h option

The full version is the man page for the command, which opens up in a browser on some systems. It is invoked by using one of two forms: either adding a --help after a command or using the help command itself as in `git commit --help` or `git help commit`.

With either of these forms, you have access to the full documentation on the command and all its options, with explanations and some examples, via the man page. This is useful when you need to understand more about an option or command.

The format for the help command is as follows:

```
git help [-a|--all] [-g|--guide]
         [-i|--info|-m|--man|-w|--web] [COMMAND|GUIDE]
```

The guide part of this command refers to some brief but helpful documentation on different aspects of using Git that you can select through help.

For example, here's a command to display a built-in guide that is a glossary: `git help glossary`.

You can use the command `git help -g` to get a list of all available built-in guides. (Be aware that some of these guides might be out of date.)

The remaining options have to do with whether the help is displayed as a web page, man page, and so on. You can specify the format to use by setting the help.format setting—for example with the commands `git config --global help.format man` or `git config --global help.format web`.

USING WEB-BASED HELP ON OS X

If you are trying to get web-based help working on OS X, you may be running into a problem where the help files are always presented as man pages. Setting help .format to web on OS X will sometimes return the following error:

'/usr/local/git/share/doc/git-doc' : Not a documentation directory

To fix this, go to `/usr/local/git/share`. Create a doc subdirectory and change into it. Issue the following clone command to populate the area with the necessary files:

```
$ sudo git clone git://git.kernel.org/pub/scm/git/git-htmldocs.git
git-doc
```

Then set the help.format value to web and try again.

Figure 5-2 shows part of a web man page for one Git command.

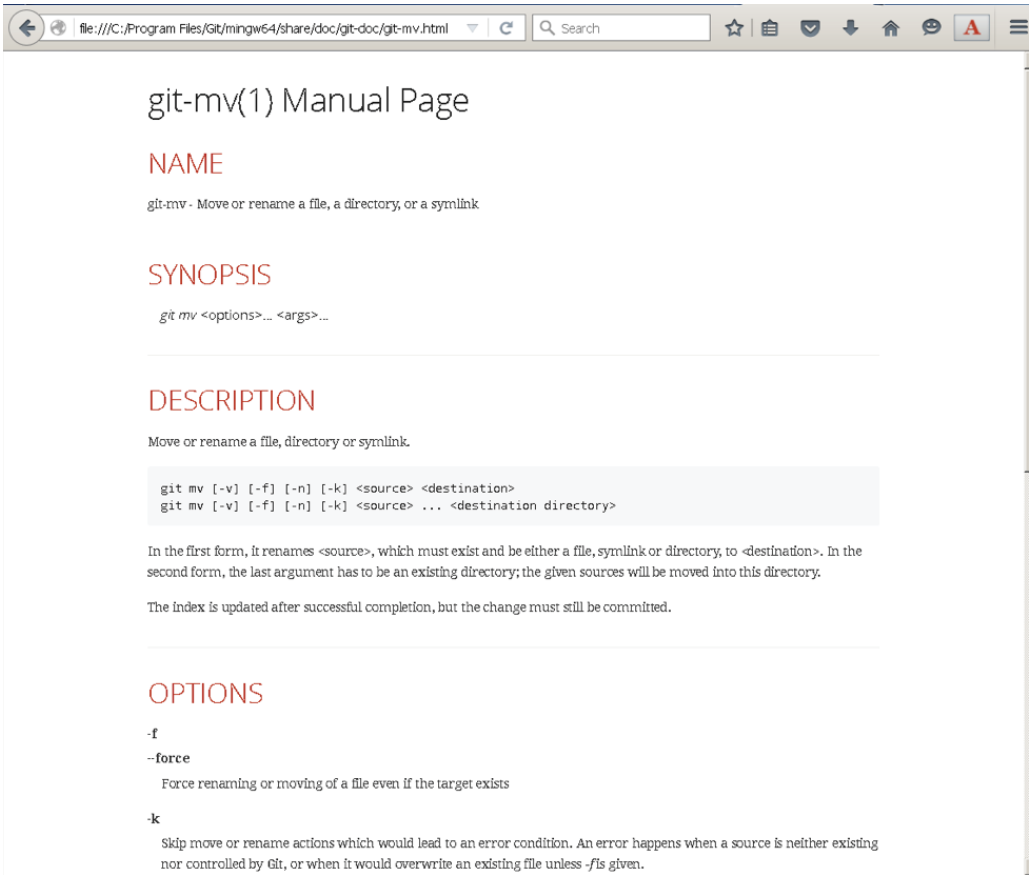


FIGURE 5-2: Git browser-based man page

Of course, you can always Google a particular command or option to find out more about it.

THE MULTIPLE REPOSITORIES MODEL

In Chapter 2, you explored several key design considerations for repositories, including repository scope and file scope. The factors I discussed there explained why Git works best with multiple, smaller repositories rather than larger, monolithic ones.

With this model, it is common to have each of the *modules* of your project housed in a different Git repository. As a result, you may need to clone several different repositories to get everything you need to work with locally. Each repository ends up in a separate directory tree on your disk.

Likewise, if you're starting a new project, you may be creating new modules that are each targeted for a separate Git repository. As I discussed in Chapter 4, the git init command is used for creating new repositories, one per directory tree.

Although working with multiple repositories at the same time is common in Git, it is a different way of working for most people. Figure 5-3 shows a diagram that represents these kinds of scenarios. Here, some repositories are newly created by the init command, and some are cloned down from existing remote repositories. Notice that each repository is housed in a separate working directory where the actual repository is physically stored in the .git subdirectory tree within that directory.

STARTING WORK WITH GIT – MULTIPLE REPOSITORIES

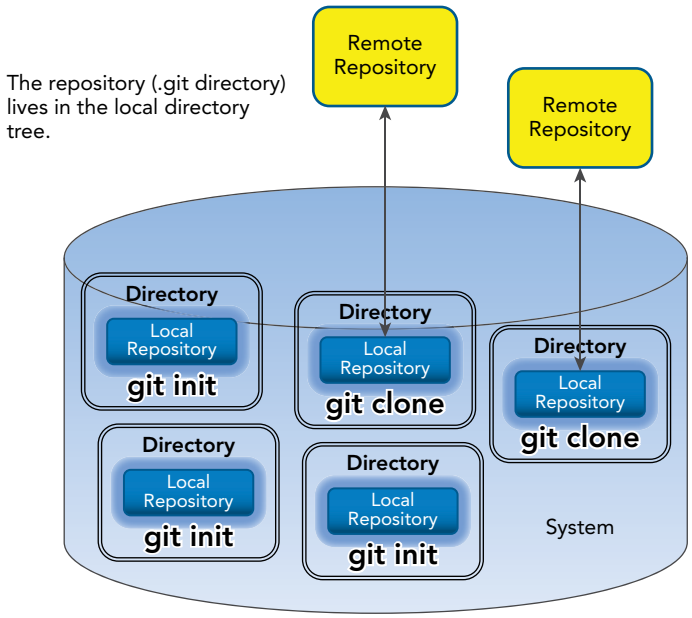


FIGURE 5-3: Working with multiple repositories

In Chapter 4, I also talked about configuration for Git and the different levels: system, global, and local. To illustrate how that would work in a multiple repository model, you could group these directories into multiple users on the system with configuration files at the appropriate levels.

Figure 5-4 shows one possible organization. Note that each repository has its own local configuration as represented by the document icons in each directory. Further, each user has their own global

configuration (for all of their repositories) as represented by the document icons in the user sections. Finally, there is one system configuration (for all users) as represented by the document icon next to the System title.

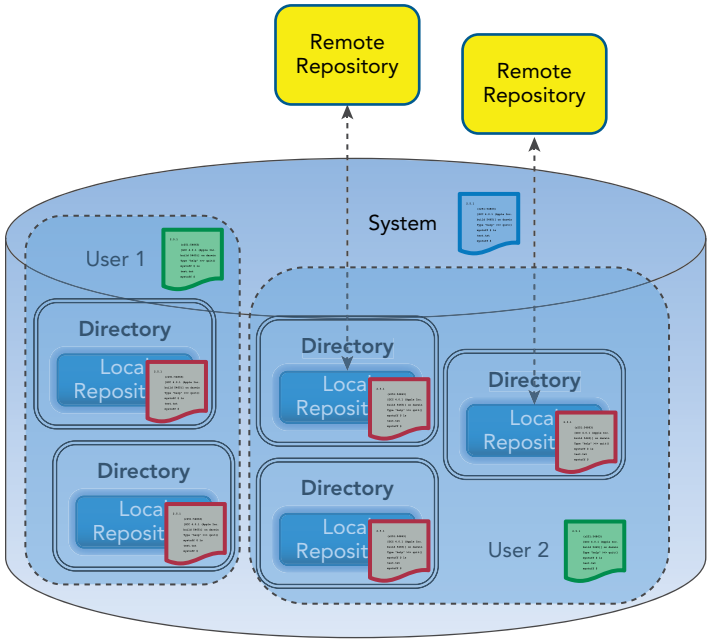


FIGURE 5-4: Overlaying configuration files on your model

ADDING CONTENT TO TRACK—ADD

I've already talked about adding content to Git with the *add* command. The dark arrow in Figure 5-5 reminds you where adding and staging fits within the overall promotion model workflow.

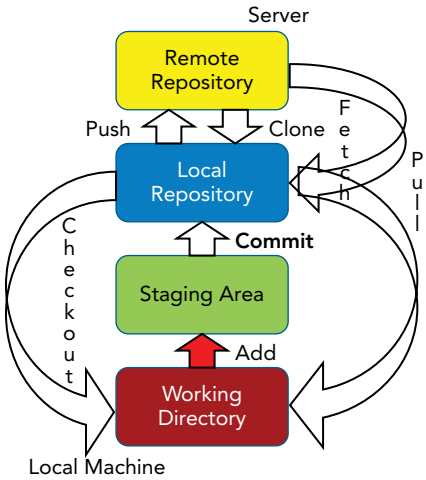


FIGURE 5-5: Where adding and staging fit in

It's worth spending a moment here to discuss what I mean by three related terms: *tracking*, *staging*, and *adding*.

Tracking refers to having Git control and monitor a file. The first step in getting Git to track a file is *staging* it. Here, staging means that you tell Git to take the latest change from your working directory and put it in the staging area. You do this using the Git add command. This is why I sometimes refer to *staging a file* as *adding a file* and vice versa.

Another important point is that whether you are staging a completely new file that is not currently tracked by Git, or staging an update to a file already tracked by Git, you still use the add command. Think of it as always *adding* content into Git.

Staging Scope

As I discussed in Chapter 3, one of the purposes of the staging area is to allow you to build up a complete set of content to commit as a unit into the local repository. When it is done in *stages* like this, a user may be staging only a subset of eligible files at a time—some files may not be ready. As an example, I could do *git add file1* followed by *git add file2* followed by *git add file **.

For users who don't choose to use the staging area in this way, it is more common to just stage everything that is eligible. The command git add . does this for you. (Note that "." is a required part of the command here.).

You can also supply a pattern to select groups of files from the directory structure, as with a command like git add *.c that selects only files with a ".c" extension.

By *everything that is eligible* above, I meant all files that are new or updated AND not ignored. *New or updated* is self-explanatory. *Not ignored* requires further explanation.

Ignoring Files

Typically, when working in a local directory tree on a project, there is some subset of files that you don't want (or need) the source management system to track. Examples include those files I talked about in Chapter 2: generated output files that should be re-created from the source each time, or external dependencies that are stored and managed in another system (such as an artifact repository).

To tell Git to ignore certain files (meaning not to track them), you just need to list them in a *Git ignore file*. This is a text file named *.gitignore* that is placed at the root (top level directory) of the local environment. If this file exists locally, Git will read it and ignore files and directories that match the names and patterns specified within it.

The Git ignore file is covered in more detail in Chapter 10. While not strictly required, having a Git ignore file is considered a best practice for any project managed by Git.

Partial Staging

Before you begin, this section outlines functionality that can be useful but is not required for using Git. If you are only interested in basic staging of files, you may want to skip over this topic for now.

On the opposite end of the spectrum from staging all eligible files or sets of files, Git includes an option that allows for *partial staging*. This means choosing to take selected changes from a file, but

not necessarily all of them. You can use the -p option to do this, as in `git add -p <file or . or pattern>`.

This command tells Git to treat the changes in any file being staged as one or more separate *hunks*. Here, a hunk is a change to a set of lines that is separated from other hunks by a set of unchanged lines. The number of hunks also depends somewhat on the size of the file. For small files, even those with several changes, Git may present the entire set of differences as a single hunk.

> **NOTE** *If you try to do a partial add for a file that has not been added to Git previously, Git will tell you there are* no changes. *You need to add a copy of the file into Git first in the standard way (not as a patch) so that there is a base there to* patch against.

Through an interface that Git presents, users can choose which hunks they want to have staged and which they don't, as well as other functionality. The interface will show the first hunk of the file, followed by a prompt. Here's a simple example of output from the add with -p option:

```
diff --git a/file b/file
index SHA1..SHA1 filemode
--- a/file
+++ b/file
@@ -1,7 +1,7 @@
line 1
line 2
line 3
-line4
+line 4
line 5
line 6
line 7

Stage this hunk [y,n,q,a,d,/,s,e,?]?
```

What do you need to know from this? It is essentially a diff between the version in Git and the version in the working directory. These are represented as *a* and *b* in the header. The line, "@@ -1,7 +1,7 @@", describes the range of differences for the two files. You can think of that line like this: *Before the changes in this hunk, designated by the "-", starting at line 1, you had 7 lines. After applying the changes in this hunk, designated by the "+", there should be 7 lines.*

In the actual listing, lines that are added show up with a "+" in front of them. Lines that are deleted show up with a "-" in front of them. In this particular case, I modified the same line, but here, Git shows it as one line being removed in the original version and another line being added in the new version. As a result, the before and after line counts are the same.

Now that you know how to interpret the hunk, you can decide what to do with it. If you select ? (or an option that isn't supported), Git will display the meaning of the different available *subcommands* as follows:

```
y - stage this hunk
n - do not stage this hunk
```

```
q - quit; do not stage this hunk or any of the remaining ones
a - stage this hunk and all later hunks in the file
d - do not stage this hunk or 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
```

Let's look at a couple of the most useful subcommands here. As implied by the help text, *y* tells Git to stage this hunk. This means that this portion of the file's changes will be staged. Likewise, selecting *n* means that this portion of the file's changes will not be staged. Essentially, you are selecting which changes you want to take from the file or files in your working directory and stage for a future commit into the repository.

Most of the other subcommands are for doing bulk operations with hunks or navigating around the set of hunks. If you select *g* and have multiple hunks, Git presents you with a list of the available hunks identified by number and allows you to select which one you want to work with next. If you type a "/" and specify text found in the file, Git will jump you to the hunk with that text.

Two other subcommands of the patch staging interface are *s* for *split* and *e* for *edit*. I'll briefly discuss the use of each one.

The split subcommand tells Git to split the file into smaller, separate hunks during an add operation with the patch option. This is useful if you have a fairly small file and Git presents it initially as one single hunk. Note that if you do not see an *s* in the prompt list, this means that Git has already split it down as small as it reasonably can. This subcommand can be useful to let you get finer-grained control to stage or not stage smaller changes instead of having to try and deal with one big change.

Editing a hunk allows you to modify the lines within it. When you choose this option, Git brings up the configured editor with the hunk in the patch format. The idea is to make your edits, save the file, and exit the editor.

Each line of a hunk is indented one space in the editor. The first column is used as a way to specify the changes to make. Based on the existing changes between the two versions of a file, lines to be added have a "+" in the first column and lines to be deleted have a "-" in the first column. To remove one of these lines, the built-in help suggests deleting the line if it has a "+" or changing the "-" to a " " if you want to remove a line starting with a "-". Other changes can be made in the patch, but they will increase the probability of the problems I'll talk about next.

Figure 5-6 shows an example of a session for editing a hunk.

The Problems with Editing Hunks

Editing hunks via the Git command line is not recommended for beginners. The reason for this is that you are essentially editing a patch to be applied against a file. However, this patch is based on a starting place in the file and an expected number of lines (that is, the information between the @@ signs in the header).

```
# Manual hunk edit mode -- see bottom for a quick guide
@@ -1,6 +1,7 @@
 1
 2
 3
+4
 5
 6
 7
# ---
# To remove '-' lines, make them ' ' lines (context).
# To remove '+' lines, delete them.
# Lines starting with # will be removed.
#
# If the patch applies cleanly, the edited hunk will immediately be
# marked for staging. If it does not apply cleanly, you will be given
# an opportunity to edit again. If all lines of the hunk are removed,
# then the edit is aborted and the hunk is left unchanged.
```

FIGURE 5-6: An edit session for a hunk

It is very easy to make an edit that will cause the patch to not align with the starting line and the expected number of lines. When that happens, the patch will not apply. After you exit the editor, you will see a message that says something like this: "*Your edited hunk does not apply. Edit again (saying "no" discards!) [y/n]?*". This message may also be accompanied by an equally dubious one such as this: *"fatal: corrupt patch at line ##"*.

This means that some change you made in the editor caused the patch (this hunk) to not be able to merge into the rest of the file. This is an easy state to get into and a hard state to get out of, especially because modifications in an earlier patch can affect the expected starting line and line counts for later patches. To make this all work from the command line in all but the simplest cases requires some calculations on where a particular patch should start, the number of lines affected, and so on.

A better option is to stage those hunks that are ready, and not stage the ones that need further edits. You then edit the entire file in an editor, make the edits as needed, and stage those updated changes. (If needed and available, the split subcommand can further reduce the size of hunks before doing this.)

> **NOTE** *While many operations in the Git command line provide increased functionality versus doing the operation in a GUI, selectively editing and staging parts of a file can be simplified using a GUI interface. In this kind of interface, users can often select and update content without having to worry about the line numbers and relative locations typically associated with patches.*

Interactive Staging of Commits

There is one more variant of the staging (add) and commit functions that is available to users: interactive staging. This option presents a different command line interface that lists the various files and available staging functions and assigns a letter or number to each one. You then choose content and perform operations by entering the corresponding letters or numbers at an interactive prompt.

To invoke this function, you must add the --interactive option at the time you execute the command. Here are some examples:

```
$ git add --interactive
$ git add --interactive *.c
$ git commit -m "update" --interactive
$ git commit --interactive
$ git commit --interactive -m "my change" file1.java
```

In short, you can add the --interactive option on any add or commit command line to use this interface.

The interface actually performs the same function whether you are running it as part of an add command or a commit command—it allows control over what is in the staging area using a more concise interface.

As a brief example of how the interface works, consider a case where you have a new Git repository with three files (*file1.txt*, *file2.txt*, and *file3.txt*) that have not yet been added to Git. In this state, the files are called *untracked* files (more on that in the next chapter).

Now if you run the add command with the interactive option, you are presented with the interactive listing and prompt.

```
$ git add --interactive

*** Commands ***
  1: status       2: update       3: revert       4: add untracked
  5: patch        6: diff         7: quit         8: help
What now>
```

Notice that the prompt is asking what you want to do now. You indicate which operation by entering either the number or the first letter (highlighted) of the command from the listing. In this case, you'll add (stage) some of the currently untracked files. To do this, you start the operation by choosing 4 or *a*.

```
What now> a
  1: file1.txt
  2: file2.txt
  3: file3.txt
Add untracked>>
```

You're presented with a list of the untracked files in the directory. Each file has been assigned a number by which you can refer to it. In this case, you'll add (stage) files 1 and 3. You could do this via two separate inputs, or via a comma-separated list. Here, you'll use the latter format.

```
Add untracked>> 1,3
* 1: file1.txt
  2: file2.txt
* 3: file3.txt
```

After you do this, Git tells you that you've staged the two files by putting the "*" in front of their names. Because you're done with this command, you can just press Enter/Return with nothing after the prompt to return to the main prompt. Git tells you that two paths (files) were added.

```
Add untracked>>
added 2 paths
```

```
*** Commands ***
  1: status       2: update       3: revert       4: add untracked
  5: patch        6: diff         7: quit         8: help
What now>
```

If you now choose the status command, Git displays in this concise format what you have in the staging area and how it relates to what you have in your working directory.

```
What now> s
           staged     unstaged path
  1:        +1/-0      nothing file1.txt
  2:        +1/-0      nothing file3.txt

*** Commands ***
  1: status       2: update       3: revert       4: add untracked
  5: patch        6: diff         7: quit         8: help
```

Let's take a closer look at how to read this status for the first file.

```
             staged     unstaged path
  1:        +1/-0      nothing file1.txt
```

The number in front (*1*) is just an identifier that you can use to reference this item in the staging area if you update it further using this interface.

The numbers under *staged* represent the number of lines added since you started staging this file and the number deleted. In this instance, *file1.txt* only contained one line, so you see one line added and zero lines deleted.

Under *unstaged* you see *nothing*, which, of course, indicates that nothing is unstaged. Think of this as *what's different between the staging area and the working directory* or *what's new in the working directory for this file*. Because you don't have any changes in the file in the working directory that aren't staged, the version in the working directory and the version in the staging area are the same, so nothing is different. If there were differences, they would be in the same +/- format as used for the *staged* column.

Finally you have the path name, which, in this case, is just the filename.

Now, suppose you add a line in your working directory to file1.txt so that it has two lines instead of one. (This would be done outside of the interactive interface.) If you want to see what's different between the version you have staged and the updated one, you can use the *diff* command here.

```
What now> d
             staged     unstaged path
  1:        +1/-0        +1/-0 file1.txt
  2:        +1/-0      nothing file3.txt
Review diff>>
```

You get a summary status. Notice that the *unstaged* section now shows *+1/-0* because the staged and unstaged versions in the directory are different. The way to read this is that in the unstaged version of the file, one new line has been added (which I did previously) and no lines deleted.

Your prompt has also changed to be relative to the command you selected and to allow you to choose which file you want to diff further (if you do). If you want to look at the actual diff for the file you changed, you can input 1 and get output like the following:

```
Review diff>> 1
diff --git a/file1.txt b/file1.txt
```

```
new file mode 100644
index 0000000..257cc56
--- /dev/null
+++ b/file1.txt
@@ -0,0 +1 @@
+newline
*** Commands ***
  1: status       2: update       3: revert       4: add untracked
  5: patch        6: diff         7: quit         8: help
What now>
```

This is the same type of patch format that I talked about earlier in the section, "Partial Staging." Now, to get your updated content in the staging area, you can use the *update* command. The workflow will be as it was for the other commands.

1. You will get the same kind of list of what is eligible to update.
2. You can then select the number that corresponds to the item you want to update and you'll get the "*" marker to indicate it was done.
3. You can then just press Enter/Return to exit the update mode.

The sequence looks like this:

```
What now> 2
            staged     unstaged path
  1:        +1/-0        +1/-0 file1.txt
Update>> 1
            staged     unstaged path
* 1:        +1/-0        +1/-0 file1.txt
Update>>
updated one path

*** Commands ***
  1: status       2: update       3: revert       4: add untracked
  5: patch        6: diff         7: quit         8: help
```

If you now take a look at the status after this update, you'll see the following:

```
What now> s
            staged     unstaged path
  1:          +2/-0      nothing file1.txt
  2:          +1/-0      nothing file3.txt
```

Note that you have two lines added for the file since you started staging it. Also, you are back to nothing unstaged because all of the changes made in the working directory have been added to the staging area.

Lastly, if you decide you want to *unstage* a set of changes, you can use the revert command to do so. The sequence is the same as for the others: select the command, select the file, and the operation is executed.

```
*** Commands ***
  1: status       2: update       3: revert       4: add untracked
  5: patch        6: diff         7: quit         8: help
What now> 3
            staged     unstaged path
  1:          +2/-0      nothing file1.txt
```

```
  2:          +1/-0      nothing file3.txt
Revert>> 1
             staged     unstaged path
* 1:          +2/-0      nothing file1.txt
  2:          +1/-0      nothing file3.txt
Revert>>
rm 'file1.txt'
reverted one path
```

A status command now shows only the one file remaining in the staging area.

```
*** Commands ***
  1: status       2: update       3: revert       4: add untracked
  5: patch        6: diff         7: quit         8: help
What now> s
           staged     unstaged path
  1:        +1/-0      nothing file3.txt
```

For the remaining commands, patch will launch a similar workflow that allows for partial staging (as described in the section, "Partial Staging"). Also, as the name implies, *help* provides a quick summary of what the main commands do.

Once you quit the interactive staging process, Git provides a brief status summary.

```
What now> q
Bye.
[master 6a43d7e] update file3.txt
 1 file changed, 3 insertions(+)
```

SUMMARY OF THE INTERACTIVE STAGING WORKFLOW

Start the interactive option: *git add --interactive* (or *git add -i*)

A list of available commands appears, along with unique numbers to select each of them. You can also use the first letter of the command to select it.

A *what now>* prompt appears for input. At the prompt, enter the letter or number corresponding to the command you want to use.

The files that you can choose to operate on appear in a list, with a number to identify each one.

The prompt changes to reflect the current operation.

Select the files you want to work with by entering the individual numbers, or a comma-separated list for multiple ones, or a range of numbers separated by a hyphen.

The operation takes place on those files.

Repeat for any other files to which you want to apply the same operation.

When done with the operation, press Enter/Return (without any line numbers) at the operation prompt to return to the main *what now>* prompt.

Bypassing the Staging Area

In Chapter 3, I discussed the various uses and reasons for the staging area as a separate level in the Git promotion model. However, if you don't need to have your changes staged as a separate step in the process, there is a shortcut that Git provides—although it is qualified.

The shortcut is to use the *-am* option on the command line when doing a commit, as in *git commit -am "comment"*.

I'll talk more about the commit operation shortly, but the *-am* option effectively tells Git to stage and commit the updated content in one operation. It's a nice convenience when you don't need to hold the change in the staging area for any reason.

COMBINING OPTIONS IN GIT

I mentioned in Chapter 4 that options can be supplied to Git commands either spelled out completely (and preceded by two dashes) or abbreviated by their first letter (and preceded by a single dash)—for example, *--all* versus *-a*.

The abbreviated form of options can be combined together where it makes sense. As I mentioned, you can use *-am* to add and commit new versions of files.

Here, *-am* is a contraction of the *-a* and *-m* options: *-a* is the short version of *--all*, an option that tells Git to stage all eligible changes before doing the commit, and *-m* (short for *--message*) is used to supply the message or comment for the commit.

Combining the options in this way works because *-a* does not take an argument, while *-m* does. As a result, *-a* is interpreted as a standalone option.

Trying to combine the options in the reverse manner (*-ma*) does not work. This is because *-m* expects an argument (the commit message/comment). Specifying *-ma* causes Git to interpret the *a* part as the expected argument to *-m* and tells Git the commit message/comment for this operation is *a*—not what you intended.

So, combining abbreviated options works in Git as long as the option (or options) before the last one does not take arguments.

The one caveat with the *-am* shortcut is that it will not work for new content or files. The first time a file is added to Git, it must have the *git add* command done first.

Some IDEs will also provide a shortcut for doing the add and commit for files in their projects—for example, being able to drag and drop content to add and commit in one step.

FINALIZING CHANGES—COMMIT

After content is staged, the next step is the commit into the local repository. This is done with the *commit* command. The syntax is shown below.

```
git commit [-a | --interactive | --patch] [-s] [-v] [-u<mode>] [--amend]
           [--dry-run] [(-c | -C | --fixup | --squash) <commit>]
           [-F <file> | -m <msg>] [--reset-author] [--allow-empty]
           [--allow-empty-message] [--no-verify] [-e] [--author=<author>]
           [--date=<date>] [--cleanup=<mode>] [--[no-]status]
           [-i | -o] [-S[<keyid>]] [--] [<file>…]
```

The dark arrow in Figure 5-7 reminds you where you are in the overall promotion model.

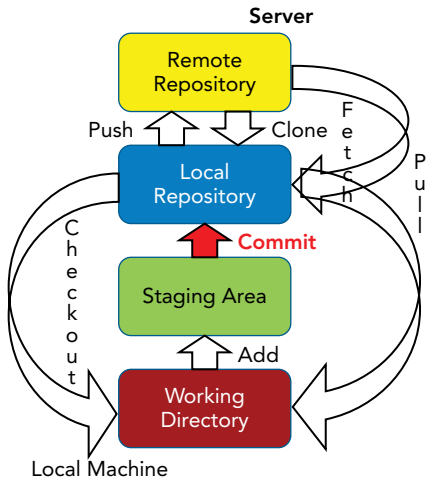


FIGURE 5-7: Where commit fits in

You can think of the commit action here as *committing to make the change permanent*. Committing always operates by *promoting* content from the staging area. (Even if you use the shortcut noted in the previous section on the commit command, you are not bypassing the staging area; you're just moving content from the working directory to the staging area and then committing it with one command.)

A key point to remember is that a commit commits changes into the local repository only. Nothing gets updated or changed in the remote repository. As I noted earlier, and as indicated in the promotion model figures, there are entirely separate commands for synchronizing content with the remote repository (discussed in Chapter 13). So, none of the changes the user commits will show up in the remote repository until those other commands are used to push them over. They are two different and distinct environments.

Prerequisites

In addition to having content in the staging area, it's best to have the username and user e-mail configured as discussed in Chapter 4. As a reminder, the commands to configure these settings on the command line are `git config --global user.name "Your Name"` and `git config --global user.email <your email address>`.

If you don't do this, then Git will attempt to figure out who you are based on the logged-in userid and the system name. If it can't, you'll be forced to set these values then. If you don't explicitly set them, then you may see a message like the following one after your first commit:

```
 [master sha1] comment
 Committer: username <username@hostname>
Your name and email address were configured automatically based
on your username and hostname. Please check that they are accurate.
```

You can suppress this message by setting them explicitly with the commands I reminded you about above. After doing this, you may fix the identity used for this commit as described in the section *Resetting the Author Information* later in this chapter.

Commit Scope

The most common form of the commit command is `git commit -m "<commit message>"`.

Here, the *-m* is the abbreviated form of the *--message* option. Git requires a message (also referred to as a comment) when doing a commit. If the commit message has spaces, it must be enclosed in quotes.

In this form, without any specific set of files or content specified, Git takes everything in the staging area and commits it. Most of the time this is what you want. (Again, the general idea is to build up a set of content in the staging area that should be committed as a unit.) However, it is also possible to commit only a selected set of content, as in `git commit -m "<commit message>" file1.c` or `git commit -m "<commit message>" *.c`.

PUTTING IT ALL TOGETHER

Figure 5-8 provides a visual way to think about the add and commit workflow. This is not exactly how things happen internally, but it is a convenient way to think about the overall process.

In part A, you start out with your local stack: local repository, staging area, and working directory. The working directory contains three files.

In part B, you specifically stage (add) one of the files, moving it into the staging area, and creating a snapshot.

In part C, you stage the remaining files by using the git add command, updating your snapshot. Recall that this form of the command (with the ".") means to traverse the directory tree, and stage all of the files that are new or changed AND not ignored (via the .gitignore file). In this case, the other two files in the working directory match these criteria, so they are staged.

Next, you commit the set of files in the staging area to create a first commit in the local repository. This is illustrated in part D. Also here, the second file is modified again in the working directory.

Now, in part E, you stage the newly modified file, creating a new snapshot, and then commit it in part F. This creates a second commit. Git is smart enough as it manages storage to not create duplicate copies of everything from the first commit, but instead *link* to it.

Amending Commits

One of the advantages and challenges I noted with Git in Chapter 1 was the ability to rewrite history. The simplest form of rewriting history in Git is *amending* the last commit. This means you are updating the last commit with content from the staging area, rather than creating a new commit with the changes.

This is done using the *--amend* option with the next commit command. The basic syntax looks like this: `git commit --amend <arguments>`.

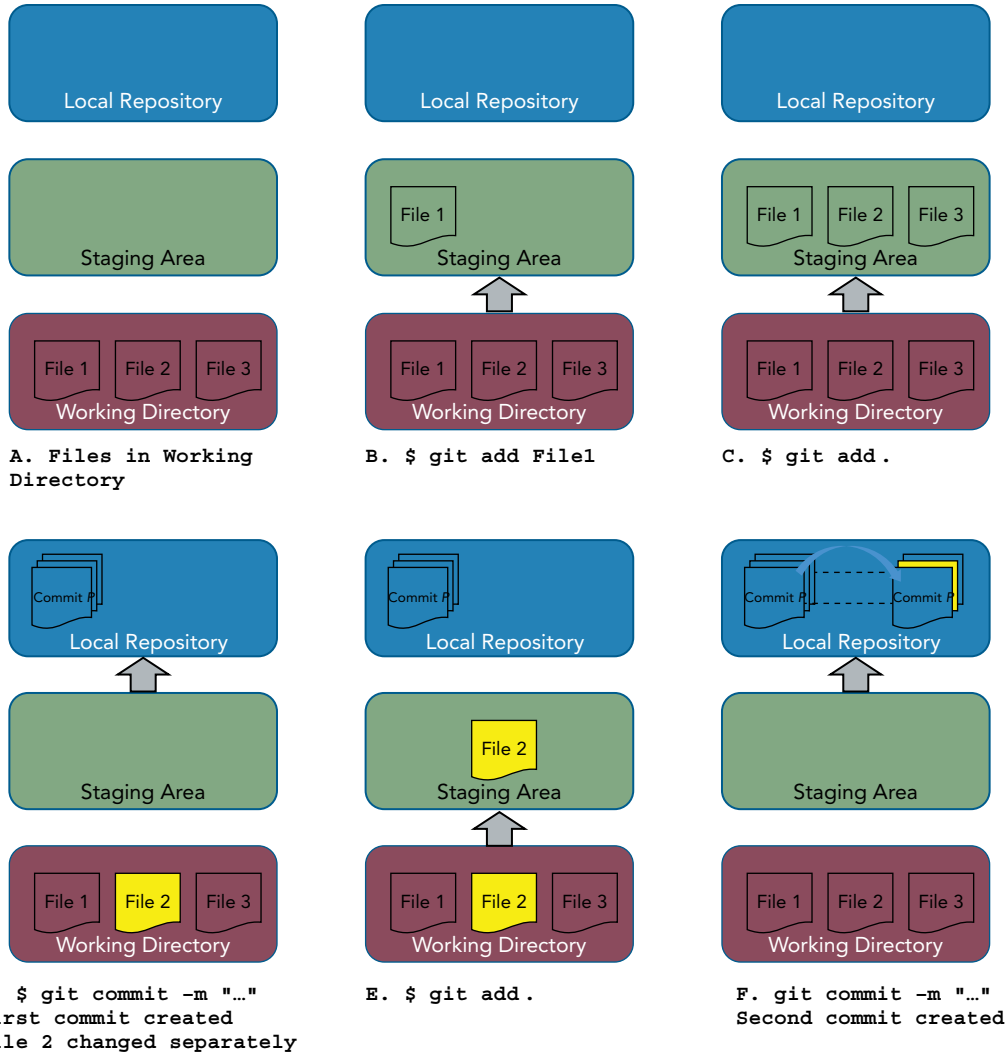


FIGURE 5-8: The basic workflow for multiple commits

Staging the Updated Content for the Amend

The amend option tells Git to update the last commit in the local repository with whatever content is currently in the staging area. If no updated content is in the staging area, then only the commit message is updated (if the user chooses to supply a new one).

Figure 5-9 shows an example of this workflow.

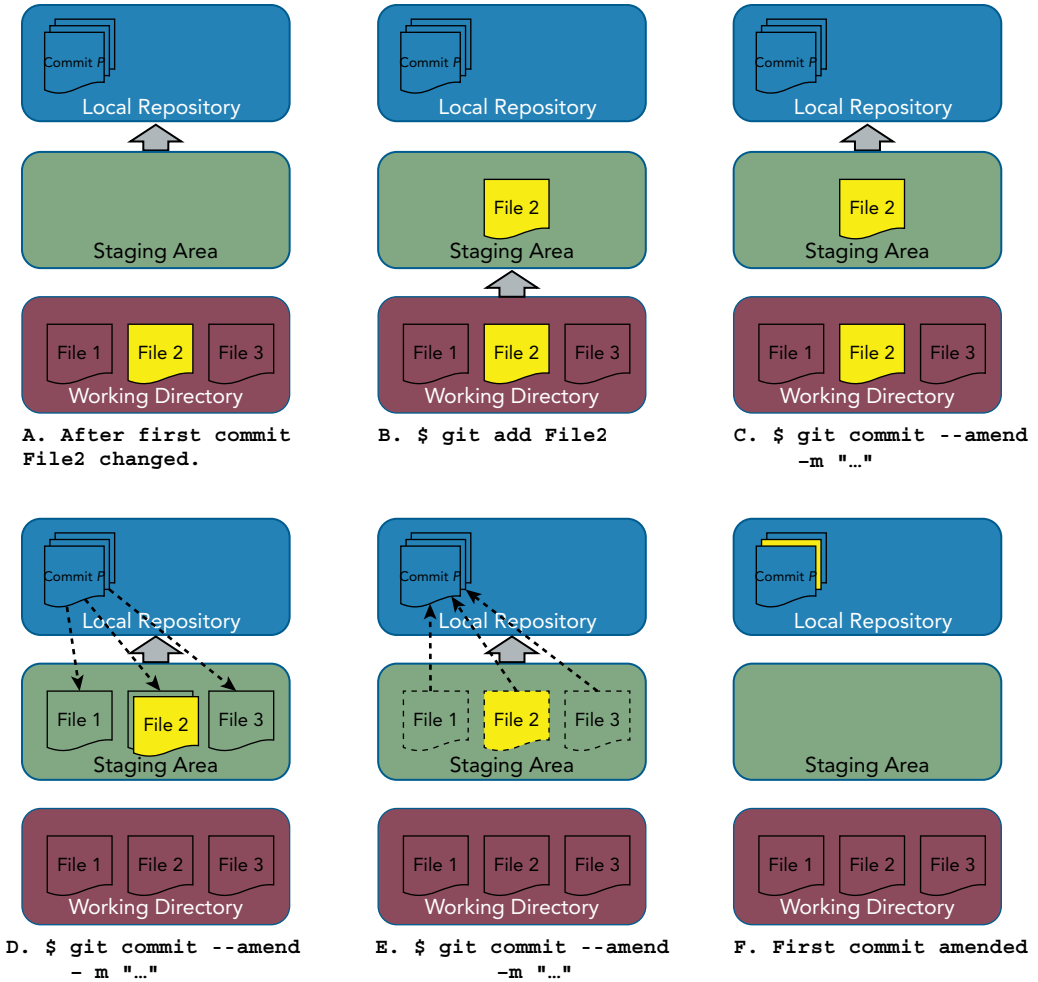


FIGURE 5-9: Workflow for an amended commit

In part A, you are starting at the point where you have one commit in the local repository and a change (in File 2) in the working directory. In part B, you are staging this change with the git add command. In part C, you commit the change, but pass the --amend option.

Instead of creating a new commit, you can think of Git pulling back the last commit (part D), expanding it, overlaying it with what's in the staging area (part E), and then updating the same commit back in the repository (part F).

Skipping the Edit of the Commit Message

While it is best practice to update the commit message when amending content, if there is a reason not to do so, you can use the --no-edit option on the amend, as in `git commit --amend --no-edit`.

Resetting the Author Information

The amend option can also come in handy if you forget to initially set the user.name or user.email configuration settings (or you have made a typo in one of them). To update the username and user e-mail captured in the previous commit, you reset the configuration settings to the desired values. You then add the --reset-author option to the commit command. After you run this command, the commit's information should show the updated values.

```
$ git commit --amend –reset-author
```

> **NOTE** *It is not recommended to amend content that has already been pushed to a remote repository where others may be working with it. Operations that rewrite history, such as amend, should ideally only be done in your local environment before content is initially pushed to the remote repository. Otherwise, other users may have accessed the copy before the rewrite and then can run into problems when they try to push their updates and discover that the history of the branch has been changed without their knowledge.*

Results of a Commit

Once a commit is executed, Git displays information like this on the command line interface:

```
$ git commit -m "add new files"

[master e3ff86b] add new files
 2 files changed, 2 insertions(+)
 create mode 100644 file1.java
 create mode 100644 file1.doc
```

I'll break down this output so you understand what Git is telling you.

On the first line, *master* refers to the default branch in Git. Until you create other branches and switch to them, you'll always be using master as your branch.

The *e3ff86b* is the first seven characters of the SHA1 value that was computed for the overall commit object—the snapshot I've referenced in previous chapters. This section is immediately followed by the commit message associated with this change.

The next line gives you information about how many files were affected by this commit, and how many changes there were in terms of insertions and deletions versus what was in the local repository before this commit.

Next, you have a list of the files that were involved in this commit along with *mode* information. The *create* text here is an indication that these are new files. The *100644* mode indicates a standard file in the repository. This is the most common mode you'll see, but other types exist for executable files, symbolic links, and so on.

GIT MODE INFORMATION

The mode information used in Git is coded as follows:

4-bit object type (Valid values in binary are 1000 [regular file], 1010 [symbolic link], and 1110 [gitlink])

3-bit unused

9-bit Unix permission (Values 0755 and 0644 are valid for regular files. Symbolic links and gitlinks have value 0 here.)

This translates into the following:

040000: Directory

100644: Regular non-executable file

100755: Regular executable file

120000: Symbolic link

160000: Gitlink. (A gitlink references a submodule commit in another repository.)

Some of these modes may show up as output from commands such as *commit*. Others may only be visible when using plumbing commands such as *cat-file* that show the modes of items in the underlying repository.

Two of these pieces of information are worth discussing in more detail: the SHA1 for the commit and the commit message.

Commit SHA1s

I discussed what a SHA1 is in Chapter 4. As a reminder, SHA1 is an acronym for Secure Hashing Algorithm 1. It is a checksum or hash that Git computes for every object it stores in its internal content management system. It is also the key that Git uses internally to map to stored content.

Whenever a commit is done in Git, Git computes a SHA1 for each piece of the snapshot that it stores (each file, directory, and so on). However, it also computes a SHA1 for the overall commit. That *commit SHA1* is the one that users see and work with. It serves as a handle or key to be able to reference that particular commit in the system. For any Git command that needs to point to a particular commit, it does that with the SHA1 value for that commit.

In terms of use, you can think of this as being similar to a version or revision number in other tracking systems—a system-generated value that identifies a particular version of a change stored in the repository.

> **NOTE** *While the SHA1 of a commit can serve a purpose similar to a revision or version number in other systems, unlike those systems, this value does not increase by some set amount each time. Rather, a SHA1 in Git is a 40-character hexadecimal string. Fortunately, you don't have to remember or specify all 40 characters—just enough to uniquely identify any commit in the system. For most systems, this turns out to be the first seven characters of the SHA1 string. For projects with significantly more commits, more characters from the SHA1 may be needed to identify a particular commit. The most I have heard of users having to specify is 12 characters. (This was for the Linux OS development where there has been a much larger number of commits over the longer period of time that Git has been in use for the development of that OS.)*

Commit Messages

When you commit into the local repository, Git requires you to supply a commit message. If you are working on the command line, you can supply one via the *-m* or *--message* argument. If you don't supply a commit message, Git will start up the default editor for your particular system for you to type in the message. Once you type in the commit message, you save the file and close the editor. The commit operation then completes.

> **NOTE** *See the "Default Editor" section in the discussion on configuration values in Chapter 4 for information on how to configure the editor for commit messages.*

When creating a commit message, it is important that it is meaningful—not just to the user doing the commit, but also to others who may be looking at it later. In general, a commit message should do the following:

- **Explain the reason for the change at a high level** (for example, refactoring xyz class, adding new foo api, fixing bug 1234, and so on). Users can use Git to see *what* was changed, but they need information to understand *why* it was changed.
- **Have a meaningful first line.** It is typical in many Git interfaces to display only the first lines of commit messages when looking at changes that have gone into the repository. For this reason, the first line should provide a brief, meaningful summary.

- **Incorporate a tracking ticket identifier** in the first line if issues are being tracked via a ticketing system. Doing this provides another reference to a place to go to get more details for users scanning the first lines of commit messages.
- **Follow any standards or guidelines** that the team or company may have for commit messages.

Chris Beams (`http://chris.beams.io/posts/git-commit/`) puts it this way:

- Separate the subject from the body with a blank line.
- Limit the subject line to 50 characters.
- Capitalize the subject line.
- Do not end the subject line with a period.
- Use the imperative mood in the subject line (for example, *fix bug 1234* rather than *fixed bug 1234*). This matches the tense used in automatic commit messages that Git generates itself for certain operations.
- Wrap the body at 72 characters.
- Use the body to explain what and why versus how.

Like well-formed comments in code, well-formed commit messages can help to ensure that you and others will find it easier to understand and maintain your changes over time. In fact, some in the Git community advocate for never using the -m option on a commit. The idea is that the -m option only suggests a short message format with less information, as opposed to always using an editor to enter the message so that more information about the commit (such as that outlined here) can be included.

ADVANCED TOPICS

In this section, you'll look at how to use templates for commit messages, as well as how to use Git's Autocorrect and Auto Execute options.

One way to help standardize commit messages and ensure good form is by using commit message templates. A commit message template is simply a text file with text and comments that suggest the type and form of content to include in the commit message. Here's an example:

```
$ cat ~/.gitmessage
Replace this line with a one-line meaningful summary


Why this change is needed:
# Explain why this change is needed


What this change accomplishes:
# Explain what this change does:

# This is our company's default commit message template.
```

```
# You should follow the following guidelines:
# Guideline 1
# Guideline 2
# Guideline 3
```

This is only one example, and obviously more could be done to make it more self-explanatory (and add real guidelines). However, this should spark some ideas. Once the template file is created, it can be saved to a global area (under the user's home directory in this example) or even to a more publicly accessible location for use among multiple users.

There are three ways for a user to tell Git to include a commit message template at the time of doing a commit:

1. Use the *-t* (--template) option on the commit command itself.

   ```
   $ git commit -t <template file location>
   ```

2. Configure the default location of the template file globally.

   ```
   $ git config --global commit.template <template file location>
   ```

3. Use a special *hook* in Git that will run at commit time. (See the section on commit hooks in Chapter 15.)

By default, information in the commit message (including from the template file) that starts with a "#" character is considered a comment and is stripped out of the commit message when the commit is actually done (as are leading and trailing whitespace and any extra blank lines).

Using the Verbose Option

Git commit includes a --verbose option. This option is designed to insert diff command output from levels in the local environment as additional information for the user while the commit message is being edited. The information is not saved as part of the final commit message; it is only inserted for the user's benefit while the commit message is being edited.

The first time this option is used on the command line, it results in the diff output between the staging area and the local repository being included. If the option is specified a second time on the same invocation, then the diff output between the working directory and staging area is also included.

The Full Commit Message Experience

Figure 5-10 shows what an editor session for the commit message looks like if you use the commit message template file described earlier and also two instances of the --verbose option. Note the parts added from your template file, added from Git, and introduced because of the --verbose options.

After the commit is done, the summary line appears in the commit output. If you look at the most recent commit message in the log, you'll see all of the text you entered. Note that the content added by Git and the comment lines that I included in the template have been stripped out.

```
$ git commit -t ~/.gitmessage --verbose --verbose
[master bc1466b] This is an example change summary line.
 1 file changed, 1 insertion(+), 1 deletion(-)

$ git log -1
```

```
commit bc1466b01125c99bb5e15f3c2242c90b923fde62
Author: User Name <email address>
Date:   Sun Apr 10 20:51:26 2016 -0400

    This is an example change summary line.

    Why this change is needed:
    The purpose of this change is to demo the commit message template file.

    What this change accomplishes:
    Updates commit message template file with example text.
```

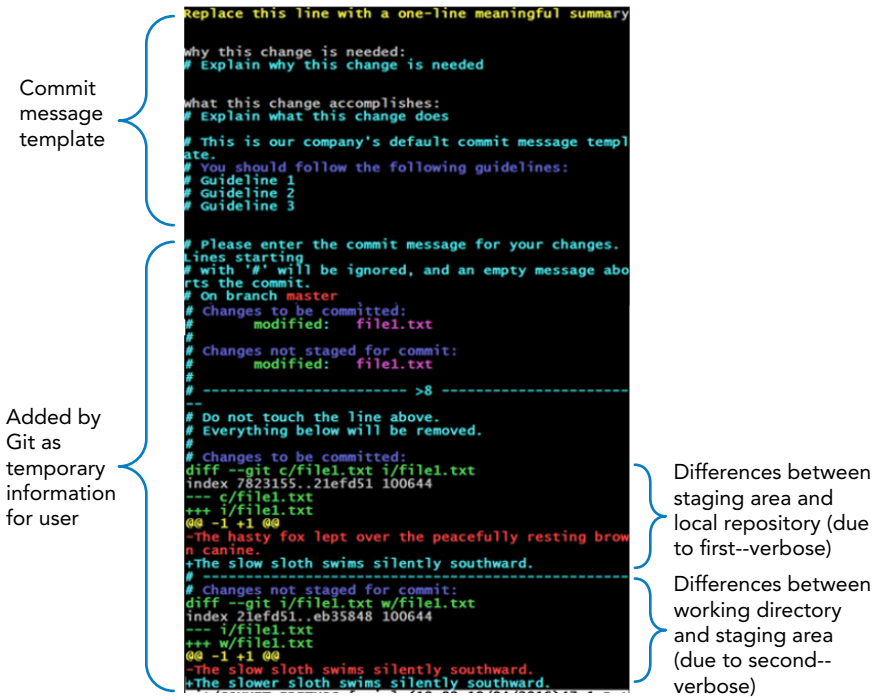


FIGURE 5-10: The editor session for a commit message using a template file and the --verbose --verbose options

Autocorrect and Auto Execute

From time to time, everyone misspells a command. Normally, Git just stops and gives you its best guess about what you meant. For example, notice what happens when you leave the *i* out of *commit*:

```
$ git commt file2.txt
git: 'commt' is not a git command. See 'git --help'.

Did you mean this?
        Commit
```

That's helpful, but what if you want to simply trust Git and have it execute the command that it thinks you intended? You can tell it to do this by changing the configuration setting *help.autocorrect* to a desired value. The value you set actually specifies an amount of time that Git waits before proceeding to execute its best guess of what you intended. The trick here is that whatever value you provide specifies a number of seconds multiplied by 0.1. So, a value of 50 means that Git waits 5 seconds to give the user a chance to cancel out before executing.

```
$ git config help.autocorrect 50

$ git commt file2.txt
WARNING: You called a Git command named 'commt', which does not exist.
Continuing under the assumption that you meant 'commit'
in 5.0 seconds automatically...
```

SUMMARY

In this chapter, you started to learn what you need to get productive using Git. I covered getting help and reinforced the multiple repositories model. From there, I dove into some details on staging changes, and described some items related to commits, including SHA1s and commit messages so you could tie the entire workflow together.

I also explained how to amend commits and how to use some advanced techniques such as commit message template files to improve commit messages.

In the next chapter, you'll take a closer look at tracking changes as they move through the Git workflow—including how to tell what content is at each level and how to see differences between the levels. Prior to reading Chapter 6, I recommend that you complete Connected Lab 2.

About Connected Lab 2: Creating and Exploring a Git Repository and Managing Content

Connected Lab 2 is your next step to reinforce the concepts covered here. It is important to work through this lab to get the hands-on experience you'll need to better understand the key concepts that will help you through the rest of this book.

The lab also includes a set of optional steps that provide more detail about the repository structure and how it evolves as content is managed in Git. This is not necessary to understand Git, but it can help you better understand how Git works.

Connected Lab 2

Creating and Exploring a Git Repository and Managing Content

In this lab, you'll create an empty Git repository on your local disk, and stage and commit content into it. You'll also explore the repository on disk to see how content is mapped logically into the physical locations.

PREREQUISITES

To complete this and all future labs in this book, you must have a working version of Git installed (2.0 or higher). If you don't have a working version of Git installed, then you should first complete Connected Lab 1: Installing Git.

OPTIONAL ADVANCED DEEP-DIVE INTO THE REPOSITORY STRUCTURE

This lab contains several optional steps, as indicated by the label at the start of each one. These steps are not needed for you to understand or use Git. They only serve to explain the underlying repository structure on disk and how it is managed in case you are interested. Feel free to skip these steps or do the deep-dive, as you see fit.

STEPS

1. On your local disk, create a new directory and change (cd) into it. (This will be the directory you work in unless otherwise specified.)
2. Initialize a new repository by running the following command:

   ```
   git init
   ```

This command creates a new git repository skeleton in a subdirectory named *.git* under the current directory—as indicated by the output message from the command. This means that you're now able to start using other Git commands in the current directory.

3. (**Optional/Deep-Dive**) If you're not interested in understanding the layout and parts of the underlying Git repository, you can skip to step 4. Understanding these components isn't necessary for you to be able to use or understand Git. However, it will help to provide a more complete mapping of how Git is managing things.

Let's take a look at what's in the repository that was just created. Open up a file/directory explorer on your system and browse into the .git directory under your current directory. (Or you can use command line commands through a terminal session.)

> **NOTE** *If you are on Windows, you may have to enable showing hidden files and directories. On the File Explorer menu bar, if there is an* Organize *drop-down menu, select it and then choose* Folder and Search Options. *Otherwise, if there is a Tools menu item, select it and then select Folder Options. Next, select the View tab and find the menu option titled* Hidden Files and Folders. *Under that option, click to select the radio button for* Show Hidden Files, Folders, and Drives.

You should see a set of files and directories there. I'll briefly discuss the purpose of each one. Again, this is just for your information, not something that you need to understand or use Git.

- **config**—This is the local configuration file where configuration settings specific to this repository are stored, as discussed in Chapter 4. If you cat the file, you see a *[core]* section with other values underneath. This file is in the INI file format.
- **description**—This file is intended for a human-readable description of this repository. It is primarily used for web browsers or some *hooks* (scripts that run before or after an operation) that want to include a description of the repository. Also, some users refer to this as the name associated with the repository.
- **HEAD**—HEAD is a pointer, or in this case, a physical file, that contains a reference to the current *item* you're pointed at in Git—usually the current commit on the current branch. Because you don't have any commits yet, this file simply contains a reference to the long form of the branch—*refs/heads/master*—because master is the default branch. (Files with this style of name are used by Gerrit to hold references to items such as SHA1 values, branch names, and so on. As the contents of the repository grow over time, more of these files will be created for specific types of references as needed.) For more information about HEADs, see Chapter 6.
- **hooks**—A hook is a script that is run before or after an operation in Git. This directory contains sample code for hooks. The filenames indicate the particular hook that would be involved if implemented. Implementation involves editing the sample code to do what is desired and putting the revised code as a file in this directory without the *.sample* extension. For more information on hooks, see Chapter 15.

- **info**—This directory is used for various internal purposes such as storing reference information for some services that require a simple way to access that. From a user perspective, it is primarily used to contain a file named *exclude*. This file is intended to be another way to tell Git to ignore certain files (not track them). It's considered another way because the primary way is using a *.gitignore* file to list those file types. The .gitignore file exists in the project and is under source management control. This is so that other users of the project can access it as well.

  See Chapter 10 for more details on the .gitignore file. On the other hand, the info/exclude file is within the .git subdirectory, and so is not managed as part of the project. This has the benefit of limiting it to only the current repository on this system. It holds exclusions (things to ignore) that are specific to this instance of this repository.

- **objects**—As the name implies, this is the directory where Git stores the internal objects it creates with the contents of commits, files, directories, and so on.

- **refs**—This directory holds the master set of references for anything that points to a particular SHA1 value, such as a branch, tag, and so on.

- **branches**—If this folder exists in your .git area, it can be ignored. It is a deprecated way to store some url information for commands that get content from the remote repository.

When you're done, switch back to your console/terminal. If you've changed out of your project directory in your terminal session, change back into the project directory.

4. Tell Git who you are by setting your basic identification configuration settings with the following commands, substituting in your name and email address as the values for the configuration. (Note the double dashes preceding *global* as you are spelling out the option. Also, values only require quotes if they contain a space.)

```
$ git config --global user.name "First-name Last-name"
$ git config --global user.email emailAddress@provider
```

5. Now let's create some content to put through the Git workflow. For the purposes of these initial labs, you just need files to work with; you don't really care what's in them. So, you can cheat and just echo something into a file using the ">" operator. In fact, the output of any command can be used to put content into a file using the ">" operator. Of course, if you prefer, you can create files using your favorite editor instead.

Create two files—their contents and names don't matter.

```
$ echo content > file1.c
$ echo content > file2.c
```

6. Stage the files with the add command. (If you prefer, you can add each separate file explicitly rather than using the ".")

```
$ git add .
```

7. (**Optional/Deep-Dive**) If you want to see how the add command changes content in your .git repository, switch back to the File Explorer or terminal and change into the .git subdirectory.

First, notice the presence of a file named *index*. In terms of Git's terminology, you can think of *stage*, *cache*, and *index* as meaning the same thing. So, this file is basically the staging area with metadata that includes SHA1 values, timestamps, and so on.

Next, look into the objects directory again. This time, you see a new two-character directory name. This is actually the start of a SHA1 value. The file inside this directoryis named with the remainder of the SHA1 value.

Change back to your project directory (the directory above .git) before continuing.

8. Commit the files using whatever comment you want.

```
$ git commit -m "comment string"
```

9. Notice the output you get. There is the branch name—the default branch—*master*, followed by an indicator that this is the first (*root*) commit and then the first few characters of the SHA1 for the commit. Take note of this value if you do the next step.

10. (**Optional/Deep-Dive**) In the File Explorer or terminal, change into the .git directory again. (Refresh the view if needed.)

Notice the *COMMIT_EDITMSG* file. This file contains the commit message from the last commit.

Go into the *objects* directory. Notice that you now have multiple subdirectories there. Each of these subdirectories starts with two characters. Find the one that starts with the same two characters as the checksum from the commit output in the last step.

Go into that directory and note that the object within has a name consisting of the rest of the SHA1 value. This is the way that Git stores objects—with the first two characters as directory names, and objects within named with the rest of the SHA1 value.

At later points you may see multiple SHA1s here in the filesystem. Git stores objects for files, directories, and commits. The one for the commit is the only one you're really interested in or will use.

Switch back to your terminal (command line) session and change back to your project directory before continuing.

11. Edit one of the files. (You can just use the ">>" to append something to the file's content.)

```
$ echo more >> file1.c
```

12. Stage and commit the file with the shortcut, using whatever text you want for the commit message.

```
$ git commit –am "comment string"
```

Take note of the SHA1 returned in the commit message if you want to do the following optional step.

13. (**Optional/Deep-Dive**) If you'd like to take a closer look at how Git maps SHA1 values to contents, you can use the plumbing command, *cat-file*. You use two options here:

    -t = type—shows the type of the object

    -p = pretty—prints information about the object

Let's start by finding the type of the SHA1 returned in the commit message output from step 12. (This is the 7 characters after "[master ".) Execute the following command:

```
$ git cat-file -t <sha1 value from the commit output>
```

You get a message back that simply says *commit*, indicating this was a commit type of object.

Now, let's print out the contents of that object. Run the command again, but with -p instead of -t.

```
$ git cat-file -p <sha1 value from the commit output>
```

This is essentially a dump of the content of the commit. Take note of the *tree* line.

So, you've looked into the commit and found the tree object. Now note the first seven characters of the SHA1 in the line starting with *tree*. Let's see what's inside the tree.

```
$ git cat-file -p <first seven characters of the SHA1 from the "tree" line>
```

Now, if you added two files as suggested, you see two filenames along with the corresponding SHA1 values listed. (The numbers on the front are *filemodes*—discussed in Chapter 6.)

Take a look at what's in one of these files. Pick the first seven characters from one of the SHA1s.

```
$ git cat-file -p <first seven characters of a SHA1 corresponding to one of the
files>
```

You now see that file's contents displayed.

14. (**Optional/Deep-Dive**) The text mentions that Git sometimes further compresses files to be more efficient. If you want to see this in practice, run the following Git garbage collection command:

```
$ git gc
```

Now, in your File Explorer or terminal, take a look in the .git subdirectory. You see a file there named packed-refs. Git combined branch and tag refs into this file as part of the compression. Now look at the *objects* directory under *.git*. Under the *info* subdirectory, you see a *packs* file. This is just a text file pointing to a *packed* file that holds compressed repository contents.

Now, if you look in the objects/*pack* subdirectory, you can see the *packed* file (.pack) that was referenced as well as an *index* file (.idx) to point to specific contents.

6

Tracking Changes

WHAT'S IN THIS CHAPTER?

- Using the Git Status and Diff commands
- Understanding tracked versus untracked files
- The meaning of HEAD and Cached
- Tricks and tips for doing different kinds of diffs

In the last chapter, you learned how to get your content into Git and how to move it through the different levels (working directory, staging area, local repository). Because you can have different versions of files at the different levels in Git, you need a way to keep track of where everything is and how the versions at the different levels may differ from each other. In short, you need easy ways to keep track of all of your work that's in progress. Git has two commands that can help you with this: status and diff. Using these two commands allows users to quickly understand the state of their changes in the local environment and to ensure that the correct changes are tracked and stored in Git.

GIT STATUS

As the name implies, the git status command provides status information on changes in the local environment that have not been committed yet. Let's start with a quick look at the general form of the command.

```
git status [<options>...] [--] [<pathspec>...]
```

Like other commands, this command can take path specifications, but those are not required. The "--" is a separator used to note where options end and path specifications start. It sits in between and is not required if the specifications are unambiguous enough.

> **NOTE** *For most of the examples, I will not include a commit, or path names and specifications, on the git status command, but they can be added if needed.*

For files that are in the working directory or staging area, the status command answers three questions: whether or not a file is tracked, what is in the staging area, and whether or not a file is modified.

Is the File Tracked or Untracked? This designation refers to whether or not Git knows about the file—that is, has someone previously added this file to Git? If this file has at least been added to the staging area at some point (and not removed), then Git knows about it, and is managing a version of it, so it is *tracked* by Git. Otherwise, the file is *untracked*—Git doesn't know about it and isn't managing any versions of it.

An example of an untracked file would be a new file that hasn't been added to Git. Files in the .gitignore file do not count as untracked because they are ignored by Git.

Git can report the status of untracked files in a couple of different ways, depending on whether or not something is staged. This brings up the next question.

What Is in the Staging Area? For this question, you are interested in whether there is anything that has been *staged* (put into the staging area via *git add*), and if so, which versions of which files. For these criteria, you'll need to understand some of the more specific terminology that Git uses to describe the status of files. I'll cover that in the example workflow shortly.

Is a Particular File Modified or Unmodified? For this designation, I am talking about whether a file in the working area is the same as, or different from, the latest version in Git. Think of *modified* here as simply meaning *different*. If it is the same, then it is *not different*, or *un-modified*. If it is different, that implies that the version in the working directory has been changed (*modified*) since it was last updated in Git.

Workflow Example with Status

To better understand file status, let's look at an example of staging and committing multiple versions of a file with its status at each step. Figure 6-1 shows a typical example of the local Git environment with the working directory, staging area, and local repository levels.

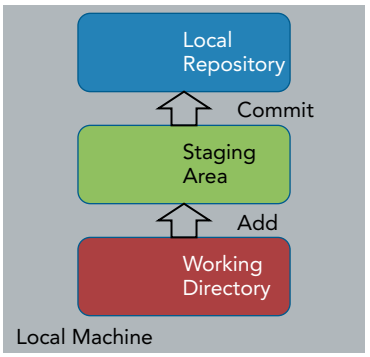


FIGURE 6-1: Empty local environment levels.

Initially, when you start with an empty directory (for example, just after a *git init* command), issuing a *git status* command will provide a message like this:

```
$ git status
On branch master
Initial commit
nothing to commit (create/copy files and use "git
add" to track)
```

This is telling you that you are on the *master* branch (the default branch in Git) and there are no changes (files) eligible to commit. Let's execute a few further steps to migrate content through the promotion model.

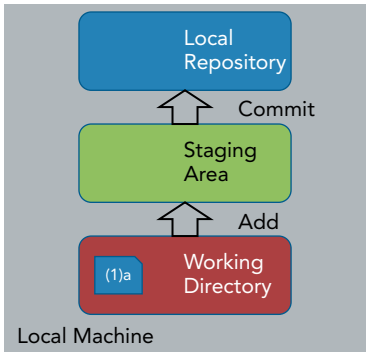


FIGURE 6-2: File created in working directory

Step 1: You create a file in the working directory with the command `echo new > file1.txt`. But you don't stage it yet. Your local environment looks like Figure 6-2. The "(1)a" notation on the file indicates this is your first step in the workflow and the file is at version *a*.

Let's look at the three questions to determine how Git sees the status:

1. **Is Git aware of the file?** No, you haven't done anything to make Git aware of it. (That is, you haven't done a git add command to tell Git about the file.) Because Git doesn't know about the file yet, it is *untracked*.
2. **What's in the staging area?** Nothing yet—you haven't done a git add command.
3. **What is the relationship of the version in the working directory to the latest version in Git?** Currently, there isn't a version *in Git*. (The only version exists in the working directory.) So, this one doesn't really apply.

Issuing the *git status* command, you'll see something like this:

```
$ git status
On branch master
Initial commit
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        file1.txt

nothing added to commit but untracked files present (use "git add" to track)
```

Note that it tells you there's nothing added to commit, meaning you haven't staged anything so there's nothing to commit (promote) to the local repository yet. It also notes the new untracked file that Git doesn't know about yet.

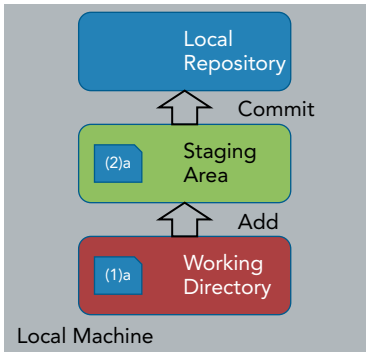


FIGURE 6-3: Version *a* of the file is staged.

Step 2: If you now stage the file with the command *git add*, your local environment looks like Figure 6-3.

Looking at the three questions:

1. **Is Git aware of the file?** Yes, because you've done a command that tells Git about it—that is, the *git add* command. The file is now considered tracked by Git.
2. **What's in the staging area?** Revision *a*. Notice that it is perfectly valid in Git to have the same revision of a file existing in multiple levels.

3. **What is the relationship of the version in the working directory to the latest version in Git?** The version in Git (in the staging area) is the same as the version in the working directory, so it is not different (not modified). Thus, it is unmodified.

When you run the status command *git status*, Git will list the new file in the staging area.

```
On branch master
Initial commit
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   file1.txt
```

Notice the terminology here of *Changes to be committed*. Whenever you see this phrase, Git is telling you about things in the staging area. You can read the *to be committed* part as indicating the next level in your promotion model. There are changes where the next level is committing into the local repository.

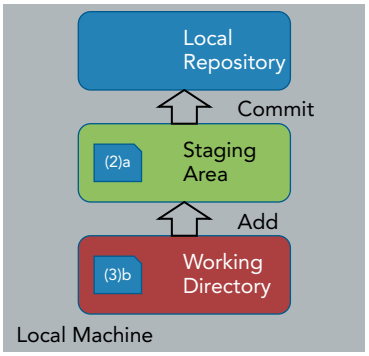


FIGURE 6-4: Update made to working directory version

Step 3: An update is made to the version of the file in the working directory using the command `echo update >> file1.txt`. The result is shown in Figure 6-4.

Looking at the questions:

1. **Is Git aware of the file?** Yes, nothing has changed for this one. You added it in Git and haven't removed it, so it is still tracked.
2. **What's in the staging area?** Revision *a*. Nothing's changed—no additional add or commit commands have been done.
3. **What is the relationship of the version in the working directory to the latest version in Git?** The version in Git (in the staging area) is still the previous version (a). The version in the working directory has been updated to a new version (b). Thus, the file is different, so it is modified.

The output from running *git status* now looks like this:

```
On branch master
Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   file1.txt

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:   file1.txt
```

This is an interesting one because it shows the filename listed twice. This can be confusing, especially if you're new to Git. The simple explanation here is that you have two different versions of the

file in two different levels of Git. Version *a* is still in the staging area, and the new version (b) is in the working directory. So, Git is reporting status on the different versions in the two levels.

Notice Git's terminology here for the two versions. The one in the staging area is listed as *Changes to be committed*. Again, you can think of this as indicating the next level; the next step in *promoting* this change is to commit it into the local repository. The version in the working directory is listed as *Changes not staged for commit*. The next step in promoting this one would be to stage it, as it is *not staged* yet.

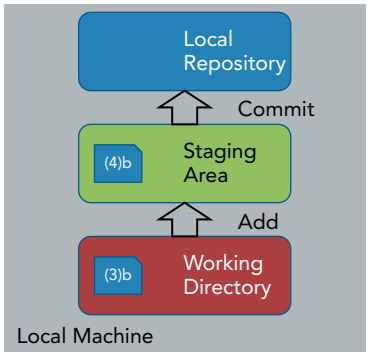


FIGURE 6-5: Version *b* staged

Step 4: Stage the new version from the working directory into the staging area. This is done with *git add*. The results are shown in Figure 6-5.

1. **Is Git aware of the file?** Yes, nothing has changed for this one.
2. **What's in the staging area?** Version *b*. Notice that version *b* overwrote version *a* in the staging area. There can only be one version of a file in the staging area.
3. **What is the relationship of the version in the working directory to the latest version in Git?** The version in Git (in the staging area) is *b*—the same as the version in the working directory. So, you're back to a status of *unmodified* (not different) for this part.

After running *git status*, the output looks the same as for Step 2.

```
On branch master
Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   file1.txt
```

Step 5: If you now commit the file with `git commit -m "new file"`, you'll see the output that I described in my discussion on commits in Chapter 5.

```
[master (root-commit) 8f8da3e] new file
1 file changed, 1 insertion(+)
create mode 100644 file1.txt
```

Your local environment looks like Figure 6-6.

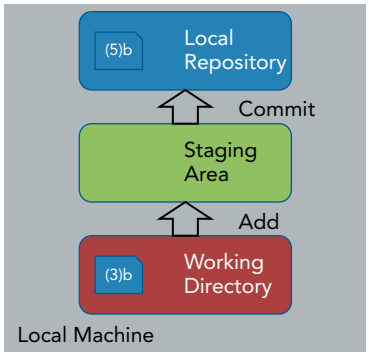


FIGURE 6-6: The file is committed.

The answers to the questions can be fairly simple at this point.

1. **Is Git aware of the file?** Yes.
2. **What's in the staging area?** Nothing; you've committed what was in the staging area.
3. **What is the relationship of the version in the working directory to the latest version in Git?** The version in Git (in the local repository now) is *b*—the same as the version in the working directory. So, it's unmodified.

Following that with a *git status* command, Git will tell you that there is *nothing to commit, working directory clean*. This is another bit of terminology that Git uses. It means that Git has the latest version of everything in its local repository that's eligible to be committed from the working directory. Basically, the local repository and working directory are in sync. There are no new changes or untracked files in the working directory that are eligible to update in Git.

Status Command Short Form

Up until now, you've been running the *git status* command in its default form. This form displays a verbose listing of status information. As you become familiar with using the status command, you may want to see more concise output. For this, you can use `git status -s`, which is the short form of the command.

The -s option causes Git to display a simpler format: one line of status per file. It displays a one- or two-character code preceding the filename to indicate the status. In most cases, you can think of the first character (from left to right) as indicating the status of the file in the staging area, and the second character as representing the status of the file in the working directory if different.

Table 6-1 lists the common status values for commands you have been working with and the representative codes.

TABLE 6-1: Git Status Codes for Short Options

STATUS	COLUMN 1 CODE	COLUMN 2 CODE
Empty working directory	blank	blank
File staged and unmodified	A	blank
File staged and modified	A	M
Untracked file	?	?

Key: A = added, M = modified, ? = untracked, blank = unmodified

So, if you map the short status to the different steps you did earlier, it would look something like this. Starting out with an empty working directory and no files in Git, `git status -s` would not have anything to show.

To mirror what you did before , you can create a file using the command `echo version1 > file1.txt`. After this, `git status -s` shows the untracked file with the output "`?? file1.txt`".

In step 2, you staged the file via `git add file1.txt`. The status shows the file as added (another word for *staged*) via the A character in the first column.

```
$ git status -s
A  file1.txt
```

In step 3, you updated the version of the file in the working directory (`echo version2 > file1.txt`). Status now has the two versions to report on: the staged version (indicated by the A) and the modified version in the working directory (indicated by the M).

```
$ git status -s
AM file1.txt
```

Step 4 used `git add .` to stage the updated version over the top of the previously staged version. So, the status is back to just having one version staged (the A), and the version in the working directory is the same as that version (indicated by the blank second column).

```
$ git status -s
A  file1.txt
```

Finally, step 5 brought you to the point of committing the file (`git commit -m "first file"`). Because Git has the same versions of everything that exists in the working directory—that is, *Working directory clean*—there is no status information to report on and thus no output from the short form of the command.

```
$ git status -s
```

> **NOTE** *It's worth noting here that there are a number of additional options and status values and codes that the status command can use and return. For now, I'm just covering the ones that pertain to the commands I'm using so far.*

GIT DIFF

In addition to git status, the other operation that allows you to have a full picture across your local environment is *git diff*. As the name implies, this command shows differences between content at the different levels in your local environment.

Let's first look at the general forms of the command:

```
git diff [options] [<commit>] [--] [<path>...]
git diff [options] --cached [<commit>] [--] [<path>...]
git diff [options] <commit> <commit> [--] [<path>...]
git diff [options] [--no-index] [--] <path> <path>
```

You'll learn about several of these forms as you go along. This command can take commits, paths, or both as arguments. If a commit is specified, that refers to an entire snapshot, and can be further qualified with paths to particular files in the snapshot if needed.

> **NOTE** *Note that for most of the examples, I will not include a commit or path names and specifications on the git diff command, but they could be added if needed.*

To fully understand this command, you first need to understand a couple of symbolic names that Git uses.

Important Symbolic Names in Git

Git uses symbolic names to refer to various items or commits. The most common one is *HEAD*. HEAD generally refers to the latest commit on the current branch, and if you think of it that way, you'll be well served. It's a shortcut for the most current thing in the repository in the branch you're working in now.

HEAD is actually a pointer or reference to a SHA1—the SHA1 for the latest commit on the current branch. HEAD is used extensively when working with Git, especially as a point to reference other commits from.

I previously discussed the staging area. *Cache* and *index* are two other terms used to reference this area; both are legacy terms in Git, and have now been replaced by the staging area terminology. So, when working with the staging area and supplying options, you may be expected to supply one of these legacy terms as the option. For all intents and purposes, you can think of *staging area*, *cache*, and *index* as referring to the same level in Git.

How to Think about Git's Approach to Diffing

From time to time in this book, I present concepts in a visual way to explain them, even though that may not be exactly how things work internally. One way to think about how Git compares things when performing a basic diff operation is by thinking about going up the promotion model to find content to diff against. Let's walk through a set of steps, as you did with the status examples, to see how this works.

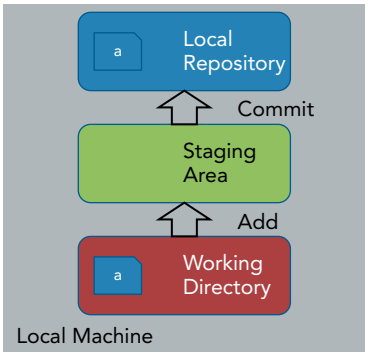


FIGURE 6-7: Starting point for diffing—working directory clean

In Figure 6-7, you pick up where you left off in the status example. You have a file at version *a* in the working directory that has been staged and committed so that you also have revision *a* in the repository. (Again, letters refer to versions as you go through the workflow.)

Step 1: First, let's do a diff between the working directory version and the current version in Git. In the working directory, you issue a git diff command. As a convenient representation, you can think of the command as starting in the working directory, and looking up to the next level (the staging area) to see if there is anything to compare against there (Figure 6-8). In this case there isn't, so it continues to look up to the next level, and finds the revision in the local repository (Figure 6-9). Comparing the version in the working directory against the one in the local repository, Git finds that they are the same.

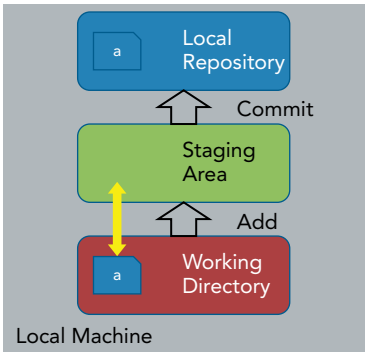


FIGURE 6-8: Workflow of git diff between working directory and Git (checking the staging area)

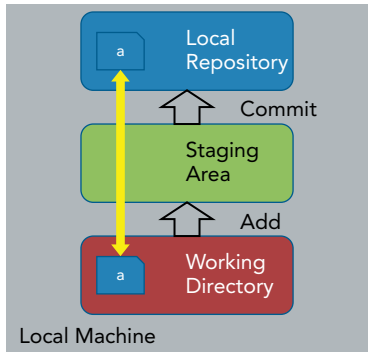


FIGURE 6-9: Workflow of git diff between working directory and Git (checking the local repository)

Command-line Git is less than user-friendly in its output when the two things it's comparing have no differences. It just returns nothing—no output messages. So, in this case, the output from running *git diff* on the command line is nothing—indicating no differences.

Suppose you now update the local file's version to *b* (Figure 6-10) and run the diff. Afterward, it continues to search up the chain until it finds the one in the local repository (Figure 6-12).

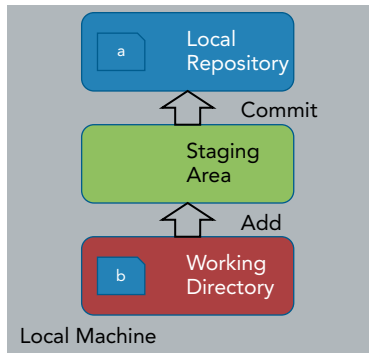


FIGURE 6-10: Local version updated to *b*

You can think of the workflow occurring in the same way again: Git starts at the working directory, searches up the chain for a version, and doesn't find one in the staging area (Figure 6-11).

Just as before, finding nothing in the staging area, you can think of Git as continuing up to the next level, where it finds the version in the local repository to diff against.

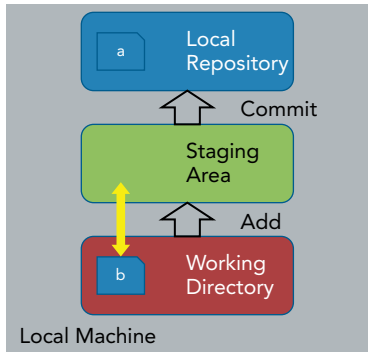


FIGURE 6-11: Diff between modified local version and Git

This time they are different, so by default, *git diff* shows the differences in *patch format*. Patch format means displaying the lines that are different, added, or deleted between the two versions. The output from the diff looks like the following:

```
diff --git a/file1.txt b/file1.txt
index df7af2c..126b36c 100644
--- a/file1.txt
+++ b/file1.txt
```

```
@@ -1 +1,2 @@
 first line
+second line
```

Now, I'll go ahead and stage version *b* so that it is in both the staging area and the working directory. Version *a* is still the latest in the local repository. This is where things get interesting.

If you simply do your normal git diff command, then Git starts at the working directory and searches up the chain to find the version in the staging area (Figure 6-13).

Those two versions are the same, so a *git diff* again shows no output (no differences).

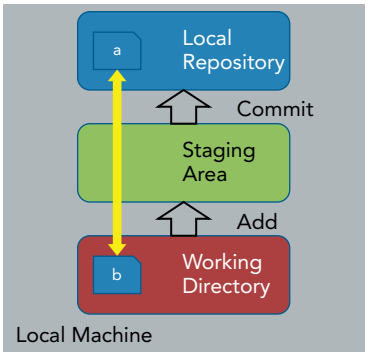


FIGURE 6-12: Diffing further up the chain

Starting at the Staging Area

The other option you may want to use when you have content in the staging area is to diff the staging area against the local repository. To do this, you add either the --cached or the --staged option to the command. (Recall that I said you can think of the *cache*, *index*, and *staging area* as the same things for your purposes.)

Adding the --cached or --staged option tells Git to start at the staging area. From there, going up the promotion levels, it compares against the local repository version. Figure 6-14 illustrates this concept.

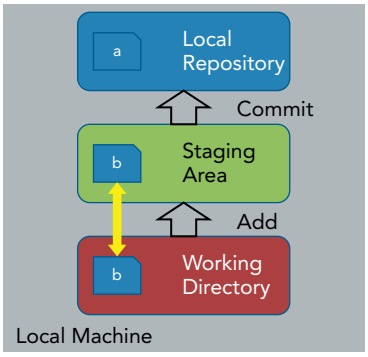


FIGURE 6-13: Diffing from the working directory with a version in the staging area

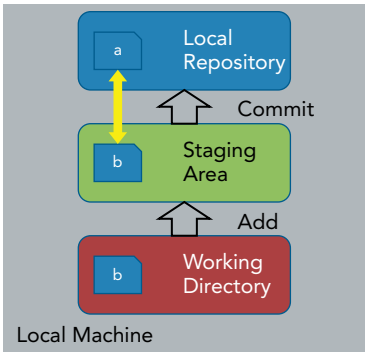


FIGURE 6-14: Diffing starting at the staging area

From the command line, you execute *git diff --staged*. The output looks like this:

```
diff --git a/file1.txt b/file1.txt
index df7af2c..126b36c 100644
--- a/file1.txt
+++ b/file1.txt
```

```
@@ -1 +1,2 @@
 version2
+second line
```

Because the version in the staging area is different from the version in the local repository (*a* is different from *b*), you see the differences.

Diffing against a Specific Version (SHA1)

One of the other useful functions of the diff command is being able to specify a revision to compare against on the diff command. The syntax is *git diff <identifier>*.

Normally the identifier will be a SHA1 (because each commit has its own unique SHA1 value) or a reference to it.

So how might this be useful? Consider a case where you might want to diff directly against the version in the local repository, bypassing the staging area. In that case, you want to diff against the latest revision in the repository. How do you specify that? Recall that I said HEAD is a pointer or reference to the latest commit (SHA1) on the current branch. So, you can use that symbolic name in your command here. If you say *git diff HEAD* then instead of going up to the staging area to check for differences, Git will bypass the staging area and compare the working directory against what's pointed to by HEAD. This would look just like Figure 6-15.

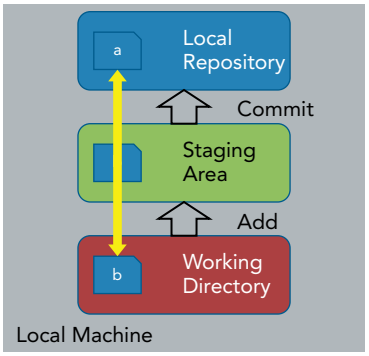


FIGURE 6-15: Diffing directly against a SHA1 (HEAD)

Again, the results would just show the differences.

```
$ git diff HEAD
diff --git a/file1.txt b/file1.txt
index df7af2c..126b36c 100644
--- a/file1.txt
+++ b/file1.txt
@@ -1 +1,2 @@
 version2
+second line
```

Diff Names Only

Like the status command, there are shorter versions of the diff output. If you only want to see the names of the files that are different, you can use the --name-only option. Running `git diff --name-only` produces the output `"file1.txt"`.

If you want to get a quick summary of which files are different and their shorthand status information, you can use the --name-status option. Running `git diff --name-status` produces the output `"M    file1.txt"`

Here the *M* means that the local version of the file has been modified—just as with the shorthand version of the status command that I covered earlier in the chapter.

Word-diff

Another option that may be useful is turning on differences at the granularity of words. You can do this with the command `git diff --word-diff`.

This will tell the diff to show differences at the granularity of *words* where words are tokenized by whitespace. Here's an example of output from the command.

```
diff --git a/file1.txt b/file1.txt
index 8e2235c..7823155 100644
--- a/file1.txt
+++ b/file1.txt
@@ -1 +1 @@
The [-quick brown-]{+hasty+} fox [-jumped-]{+leapt+} over the [-lazy-]{+peacefully
resting+} brown [-dog.-]{+canine.+}
```

The items in "[- -]" indicate things that were removed in the latest version, and the items in "{+ +}" indicate things that were added. Additional options are available for changing the way words are represented and tokenized (parsed) if desired or needed. See the help for git diff for more information on these options.

Ignoring Non-critical Changes

Obviously, the definition of what is non-critical or critical will vary from user to user and case to case. However, there is a set of common areas that can be useful to ignore when doing diffs.

Whitespace Changes

Git has a number of options to help with ignoring whitespace changes. The options are as follows:

-w | --ignore-all-space. This tells Git to ignore all whitespace changes when comparing lines.

--ignore-space-at-eol. This tells Git to ignore whitespace changes at the ends of lines.

--ignore-blank-lines. This tells Git to ignore changes in lines that are all blank.

-b | --ignore-space-change. This tells Git to ignore changes at the ends of lines and treat corresponding areas of whitespace changes as equivalent, regardless of whether they have the same amount of whitespace.

Let's look at a couple of examples. You first create a file using a simple echo command to dump the string *abcdef* into a file: `echo "abcdef" > file1.txt`. Then you stage it to get it under Git's control: *git add .*

Next, you update the file locally to have some whitespace changes (in the middle and at the end): `echo "abc def " > file1.txt`

A regular *git diff* shows differences as follows:

```
diff --git a/file1.txt b/file1.txt
index 0373d93..b6cdfe4 100644
--- a/file1.txt
+++ b/file1.txt
@@ -1 +1 @@
-abcdef
+abc def
```

Adding the option to ignore all whitespace changes (`git diff -w`) shows no differences because the only changes to the content were the addition of whitespace.

Telling Git to ignore only the whitespace at the end of each line shows the diff with the whitespace changes in the middle: `git diff --ignore-space-at-eol`

```
diff --git a/file1.txt b/file1.txt
index 0373d93..b6cdfe4 100644
--- a/file1.txt
+++ b/file1.txt
@@ -1 +1 @@
-abcdef
+abc def
```

Next, I update the local copy of the file in the working directory to only have whitespace changes at the end of each line: `echo "abcdef " > file1.txt`. If I diff it again with the same option (`git diff --ignore-space-at-eol`), it shows no differences—as expected.

Now, suppose you stage the current change and then update the file locally to a new version that only has changes in the amount of whitespace between the two versions. This can be done with a `git add .` followed by something like `echo "abc def" > file1.txt`. Then, a *git diff* between the two versions shows the expected differences.

```
diff --git a/file1.txt b/file1.txt
index d4bbed0..f9686f5 100644
--- a/file1.txt
+++ b/file1.txt
@@ -1 +1 @@
-abc  def
+abc def
```

Adding the option to ignore changes in whitespace and the amount of whitespace (`git diff -b`) shows no difference, also as expected.

DIFFERENCE BETWEEN -b AND -w

If you're wondering what the difference is between -b and -w here, it's that -b expects there to be some whitespace change at the corresponding points in both versions of the file or files being diffed, while -w doesn't care if only one version has whitespace changes. For example, if you only had whitespace changes in one version of the file as you did earlier, the -b option would not ignore the differences, whereas the -w option would.

```
$ git diff -b
diff --git a/file1.txt b/file1.txt
index 0373d93..d4bbed0 100644
--- a/file1.txt
+++ b/file1.txt
@@ -1 +1 @@
-abcdef
+abc  def

$ git diff -w
```

However, if you had whitespace changes in both versions in the corresponding areas, the diff -b would ignore the changes as in the last example in the text.

Filemode Changes

Another area where Git can sometimes identify files as changed that you may want to ignore is the executable bit of the filemode (that is, chmod+x). You can work around this with git diff by setting the configuration value core.filemode to false.

```
$ git config --global core.filemode false
```

Or, if you need to just apply this command once for a diff, you can use the one-shot configuration setting I talked about in Chapter 4.

```
$ git -c core.filemode=false git diff
```

Diffing Two Commits

The diff command can also be used to diff two different commits in the local repository. The commits could be in the same branch, or, as you'll see later, in separate branches. The output you'll see depends on the order in which you supply the SHA1 values representing the commits.

Consider the following example where you make three sets of changes for two files, doing a commit after each change.

1. First commit

```
$ echo "line 1" > file1.txt
$ echo "first line" > file2.txt
$ git add .
$ git commit -m "change 1"
```

```
[master (root-commit) c25a62d] change 1
 2 files changed, 2 insertions(+)
 create mode 100644 file1.txt
 create mode 100644 file2.txt
```

2. Second commit

```
$ echo "line 2" >> file1.txt
$ echo "second line" >> file2.txt
$ git commit -am "change 2"
[master 965b004] change 2
 2 files changed, 2 insertions(+)
```

3. Third commit

```
$ echo "line 3" >> file1.txt
$ echo "third line" >> file2.txt
$ git commit -am "change 3"
[master fc5c99f] change 3
 2 files changed, 2 insertions(+)
```

Noting the SHA1s of the first and last commits, you can now do some interesting diffing.

```
$ git diff c25a62d fc5c99f

diff --git a/file1.txt b/file1.txt
index 89b24ec..a92d664 100644
--- a/file1.txt
+++ b/file1.txt
@@ -1 +1,3 @@
 line 1
+line 2
+line 3
diff --git a/file2.txt b/file2.txt
index 08fe272..20aeba2 100644
--- a/file2.txt
+++ b/file2.txt
@@ -1 +1,3 @@
 first line
+second line
+third line
```

Here you see the differences for the two files between the first and last commits. Notice the lines that were added in the second and third commits.

Switching the order that you specify the commits gives the opposite representation: showing the lines that would be deleted to go from the most recent version to the first version.

```
$ git diff fc5c99f c25a62d
diff --git a/file1.txt b/file1.txt
index a92d664..89b24ec 100644
--- a/file1.txt
+++ b/file1.txt
@@ -1,3 +1 @@
 line 1
-line 2
-line 3
```

```
diff --git a/file2.txt b/file2.txt
index 20aeba2..08fe272 100644
--- a/file2.txt
+++ b/file2.txt
@@ -1,3 +1 @@
 first line
-second line
-third line
```

Note that you could also filter this by an individual file.

```
$ git diff fc5c99f c25a62d file2.txt
diff --git a/file2.txt b/file2.txt
index 20aeba2..08fe272 100644
--- a/file2.txt
+++ b/file2.txt
@@ -1,3 +1 @@
 first line
-second line
-third line
```

You could have also included the separator "--" between the second SHA1 and the filename, as follows:

```
$ git diff c25a62d fc5c99f -- file2.txt
```

However, that is not necessary in this case because the form of the filename is different enough from a SHA1 value to not be confused for a revision.

ALTERNATE FORMS FOR THE COMMANDS

There are several alternate forms for these commands that mean the same thing in the current context. Consider the following command:

```
$ git diff c25a62d fc5c99f
```

This can also be expressed with ".." instead of the space.

```
$ git diff c25a62d..fc5c99f
```

And, because the second commit is also the current commit on the current branch, you could use *HEAD* here as well.

```
$ git diff c25a62d HEAD
```

Finally, because the second commit is the most recent commit on the master branch, you could use *master* here instead.

```
$ git diff c25a62d master
```

Visual Diffing

As you have seen, the default presentation for the diff command is showing the changes in a standard patch format. While you are focusing primarily on the command line usage, there are times when a visual interface adds significant value or convenience. One of these cases is diffing.

Git includes a special command for working visually with differences: *difftool*. This command is actually an extended frontend for the git diff command and it can accept all the options and arguments that diff can accept.

To invoke the diffing tool, you run the command *git difftool*.

The idea is that you have one or more diffing tools installed, configured, and available for Git to use. (More about how that works in a moment.) Then, you use the git difftool command or a configuration value to select the one you want to use. The difftool command then starts up the desired tool with the appropriate arguments.

Figures 6-16 through 19 show some screenshots of several commonly used visual diffing applications (and applications that Git understands out of the box if they are installed and in the path).

FIGURE 6-16: Vimdiff

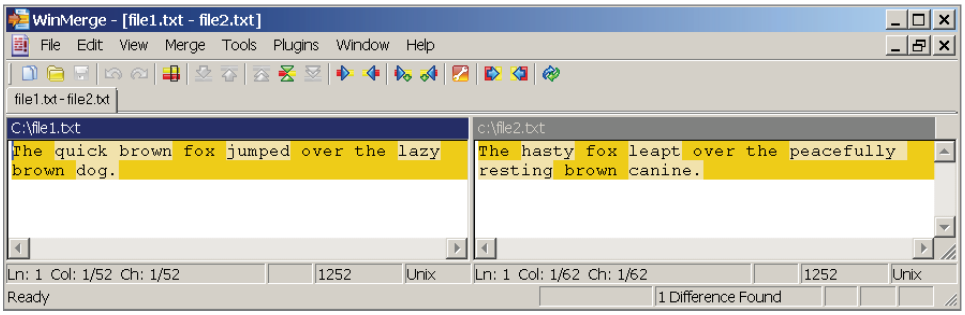


FIGURE 6-17: WinMerge

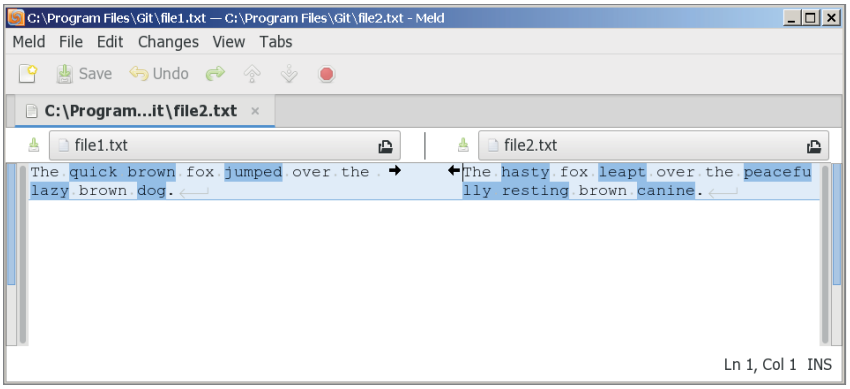


FIGURE 6-18: Meld

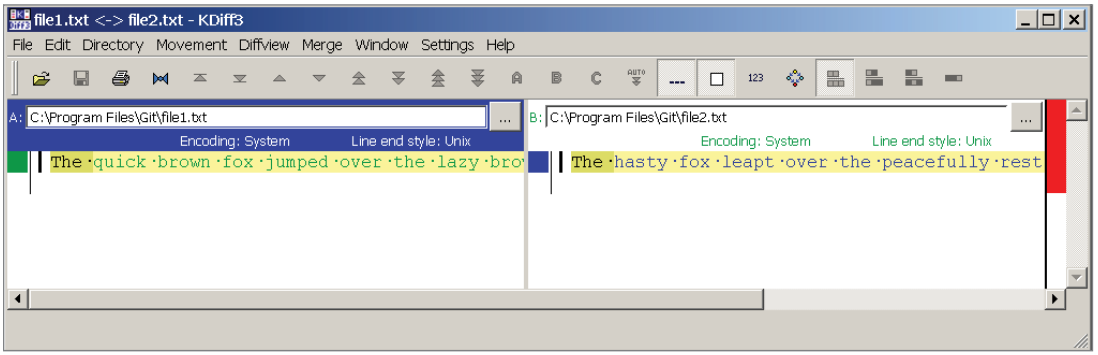


FIGURE 6-19: KDiff3

Selecting a Diffing Tool

From a list of installed, configured, and available tools, a particular tool can be selected in several different ways. However, to the difftool command, you always specify just the simple name of the tool (for example, kdiff3, vimdiff, or meld).

If no default tool has been specified, then Git will attempt to use a sensible default (usually something like vimdiff on Linux systems). One way to specify a default is to configure a particular tool via the diff.tool configuration value, as follows:

```
$ git config --global diff.tool vimdiff
```

Once this setting is configured, running *git difftool* will start that selected tool: vimdiff.

By default, Git prompts for confirmation to run the difftool for each file to be diffed. An example prompt looks like *"Viewing (1/2): 'file2.txt' Launch 'meld' [Y/n]: Y"*. This prompt can be suppressed either by supplying a no-prompt option when running difftool, as in `git difftool --no-prompt` or by setting the configuration value, difftool.prompt, to false.

```
$ git config --global difftool.prompt false
```

Another way to select a particular tool is to specify the name of the desired tool via the -t option at the time you run difftool, as in `git difftool -t meld`.

Making Diff Tools Available to Git

Git comes preconfigured to be able to work with a number of different tools for diffing. To see a list of these tools, you can run the command `git difftool --tool-help`.

Note that this does *not* mean that all of these tools are installed (or even if installed, that they can be used—they might not be in the path). What this means is that Git understands how to use these tools to do diffing without additional configuration if the tools are available on the system. The tool-help option tells you which tools are available to use (under *may be set to the following*) and which are not (under *The following tools are valid, but not currently available*).

To make one of the tools available that is marked as *not currently available*, you can install the application and make sure it is in the path. Once this is done, if it is a tool that Git knows about, it will show up in the *available* section.

If a tool is not available in the path, then you can set a configuration value named `difftool.<tool>.path` (where <*tool*> is the name of the application) to specify the location where Git can find it.

For example, Git knows how to work with an application named *Meld* for diffing when it can find it on the system. Suppose you install the Meld application on Windows in c:\meld (instead of the default Program Files location that would be in the path). To tell Git where the Meld application can be found, you would set the path value for it as follows:

(On a Windows command prompt)

```
$ git config --global difftool.meld.path c:\meld\Meld.exe
```

(On a Bash shell)

```
$ git config --global difftool.meld.path /c/meld/Meld.exe
```

> **NOTE** *The configuration of the* `difftool.<tool>.path` *value can also be used to work around differences (such as capitalization) between the actual application name and what Git expects out of the box.*
>
> *For example, Git expects Meld to be meld (lowercase). If Meld.exe is installed in a place where Git should be able to see it, it may still not be able to due to the difference in case. When you configure the tool name in* `difftool.meld.path`*, Git can then understand how to find the Meld tool.*

There is also a way to add an application that Git does not already know about. To do this, you specify a custom command to run the application in the configuration value for `difftool.<tool>.cmd`. Here again, <*tool*> is the name of the application. Git passes several variables to use with the command string. $LOCAL is set to the name of the temporary file containing the contents of the

diff before, and $REMOTE is set to the name of the temporary file containing the contents of the diff afterwards.

Other Diff Tricks

In addition to the forms of the git diff command that you have already looked at, there are a couple of others that might be useful.

Suppose that you have multiple files committed into your local repository and several of them are different. A standard invocation *git diff* may show the following:

```
diff --git i/file2.txt w/file2.txt
index ef49dd8..4ea5e4d 100644
--- i/file2.txt
+++ w/file2.txt
@@ -1 +1 @@
-more
+update
diff --git i/file3.txt w/file3.txt
index ef49dd8..4ea5e4d 100644
--- i/file3.txt
+++ w/file3.txt
@@ -1 +1 @@
-more
+update
```

An invocation of the diff command with the two files (`git diff file2.txt file3.txt`) would show the same output, assuming that file2.txt and file3.txt are the only ones that are different.

Now consider the case where you pass file1.txt and file2.txt to the command: file1.txt is not different, so there is no output for it.

```
$ git diff file1.txt file2.txt
diff --git i/file2.txt w/file2.txt
index ef49dd8..4ea5e4d 100644
--- i/file2.txt
+++ w/file2.txt
@@ -1 +1 @@
-more
+update
```

Git also supports specifying a qualified version of the file to compare against. This is similar to comparing two versions in Git. For example, to compare the version of a specific file (say, file2.txt) in the HEAD revision (the local repository's current revision) against the version in the working directory, you can use this syntax:

```
$ git diff HEAD:file2.txt file2.txt
diff --git o/file2.txt w/file2.txt
index ef49dd8..4ea5e4d 100644
--- o/file2.txt
+++ w/file2.txt
@@ -1 +1 @@
-more
+update
```

Note the *HEAD:* qualifier in front of the filename. Also, note the mnemonic prefix here of *o* for object, because you're referring to a specific object in the repository.

Even more interesting, though, is that you can use this same form to compare a qualified version of one file in Git against a completely different file locally. Here's an example:

```
$ git diff HEAD:file1.txt file3.txt
diff --git o/file3.txt w/file3.txt
index 13fd43b..4ea5e4d 100644
--- o/file3.txt
+++ w/file3.txt
@@ -1,3 +1 @@
-version2
-second line
-more
+update
```

Notice that your command actually invoked the diff against the version of file1.txt from HEAD and a completely different file (file3.txt) from the working directory. This might be useful in certain cases.

SUMMARY

In this chapter, I covered how to use two Git commands, git status and git diff, to gain a complete picture of the different content at different levels in your local Git environment. The staging area adds an extra level here that has to be taken into account. I also covered the terminology and checks that Git uses when ascertaining and reporting status. I then introduced a couple of special symbolic names or references that Git uses. Putting this all together allows you to gain a comprehensive understanding of how and where content is positioned in Git.

About Connected Lab 3: Tracking Content through the File Status Life Cycle

This Connected Lab will give you hands-on practice with the commands you've explored in this chapter, building on what you've already learned. You'll get a chance to use the status and diff commands on a project and become more familiar with Git's responses and behavior. After that, in Chapter 7, you'll move on to working with changes in Git over time.

Connected Lab 3

Tracking Content through the File Status Life Cycle

In this lab, you'll work through some simple examples of updating files in a local environment and viewing the files' status and differences between the various levels along the way.

PREREQUISITES

This lab assumes that you have done Connected Lab 2: Creating and Exploring a Git Repository and Managing Content. You should start out in the same directory as that lab.

STEPS

1. Starting in the same directory as you used for Connected Lab 2, run the status command or the short form to see how it looks when you have no changes to be staged or committed.

```
$ git status
$ git status -s
```

2. Create a new file and view the status.

```
$ echo content > file3.c
$ git status
$ git status -s
```

Question:

Is the file tracked or untracked?

Answer:

It's untracked—you haven't added the initial version to Git yet.

3. Stage the file and check its status.

```
$ git add .   (or git add file3.c)
$ git status   (git status -s if you want)
```

Questions:

a. Is the file tracked or untracked?

b. What does *Changes to be committed* mean?

Answers:

a. The file is now tracked—you've added the initial version to Git.

b. *Changes to be committed* implies that files exist in the staging area and the next step for them is to be committed into the local repository.

4. Edit the same file again in your working directory and check the status.

```
$ echo change > file3.c
$ git status
```

Questions:

a. Why do you see the file listed twice?

b. Where is the version that's listed as *Changes to be committed* (in the working directory, staging area, or local repository)?

c. Where is the version that's listed as *Changes not staged for commit* (in the working directory, staging area, or local repository)?

Answers:

a. You see the file listed twice because there is one version of the same file in the working directory and another version in the staging area.

b. The version that's listed as *Changes to be committed* is in the staging area. The phrase implies that this version's *next step* or *next level for promotion* is to the local repository using a commit.

c. The version that's listed as *Changes not staged for commit* is in the working directory. The phrase implies that this version's *next step* or *next level for promotion* is to the staging area, because it's currently *not staged*.

5. Do a diff between the version in the working directory and the version in the staging area.

```
$ git diff
```

6. Go ahead and commit and do another status check.

```
$ git commit -m "comment"
$ git status
```

Question:

Which version did you commit: the one in the staging area or the one in the working directory? (Hint: Which one is left [shows up in the status]? Note the *Changes not staged for commit* part of the status message.)

Answer:

The version in the staging area was the one committed. The content goes through the staging area and then into the local repository.

7. Stage the modified file you have in your working directory and do a status check.

```
$ git add .
$ git status
```

8. Edit the file in the working directory one more time and do a status check.

```
$ echo "change 2" > file3.c
$ git status
```

At this point, you have a version of the same file in the local repository (the one you committed in step 6), a version in the staging area (the one you staged in step 7), and a version in the working directory (step 8).

9. Diff the version in the working area against the version in the staging area.

```
$ git diff
```

10. Diff the version in the staging area against the version in the local repository.

`$git diff --staged` (or `git diff --cached`) (note that the "--" is a double hyphen)

11. Diff the version in the working directory against the version in the local repository (the one you committed earlier).

```
$ git diff HEAD
```

12. Commit using the shortcut.

```
$ git commit –am "committing another change"
```

Question:

Which version was committed (the one in the working directory or the one in the staging area)?

Answer:

Because you used the -am shortcut, the version from the working directory was staged (over the previous version in the staging area) and then that version was committed into the local repository.

13. Check the status one more time.

```
$ git status
```

Notice the output. You're back to a clean working directory—Git has the latest versions of everything you've updated.

7

Working with Changes over Time and Using Tags

WHAT'S IN THIS CHAPTER?

- Learning about the Git log command
- Filtering log output
- Filtering and searching history
- Working with Git Blame
- Tagging commits in Git
- Doing rollbacks with Git Reset
- Canceling out changes with Git Revert
- Working with signed tags
- Using reflogs

In this chapter, you'll learn about Git's history functionality, how to look at changes over time, and the many options that are available to users for displaying history information in different ways. You'll explore Git functionality that can tell you who changed each line of a file and when. Then you'll look at how to point Git back to previous versions with the Git equivalent of a rollback and how to cancel out changes. You'll also learn how to mark points in history with Git's tag functionality.

In the Advanced Topics section, you'll learn how to use signed tags for added security, and the Git reflogs functionality to track how references change over time.

THE LOG COMMAND

The key function of any source management system is tracking changes over time, as well as being able to easily identify and retrieve any previous changes. In Git, you do this with the *log* command, Git's version of a history command. The syntax is as follows:

```
git log [<options>] [<revision range>] [[--] <path>...]
```

This is a deceptively simple format description for an operation that comes with an extensive set of options, especially for putting constraints on which history items are shown and how they are presented.

With no options, the log command shows the SHA1 value, the associated e-mail address of the committer, the commit message, and any associated files in reverse chronological order (meaning newest first).

```
$ git log
commit 06efa5ecedc5db8b4834ffc0023facb70053d46e
Author: Brent Laster <bcl@nclasters.org>
Date:   Tue Sep 4 23:14:14 2012 -0400

    update field size

commit 7945579f2dcf8460dcde46c94a16e678c3113817
Author: Brent Laster <bcl@nclasters.org>
Date:   Tue Sep 4 23:13:01 2012 -0400

    update title and button text
```

Common Display and Filtering Options

While there are a large number of options that can be applied to the history output, there are some that are more frequently used in day-to-day interaction with Git. We'll cover some of those in this section.

-p. This option stands for *patch*, meaning that the history output also displays the differences, or patches, between each change. The output will look similar to the diff output you saw in Chapter 5 with the diff command.

```
$ git log --patch
commit 06efa5ecedc5db8b4834ffc0023facb70053d46e
Author: Brent Laster <bcl@nclasters.org>
Date:   Tue Sep 4 23:14:14 2012 -0400

    update field size

diff --git a/calc.html b/calc.html
index 18e5a4c..d3304b7 100644
--- a/calc.html
+++ b/calc.html
@@ -49,7 +49,7 @@ Enter a number in the first box and a number in the second box
and select the an
 <br>
```

```
 Change the operation via the dropdown selection box if desired.
 <form name="calc" action="post">
-<input type=text name=val1 size=10>
+<input type=text name=val1 size=5>

 <select name=operator>
 <option value=plus>+
@@ -62,9 +62,9 @@ Change the operation via the dropdown selection box if desired.
 <option value=avg>avg
 </select>
```

-# (where # is replaced by a number): This option means "show me the last number of commits." For example, if you only wanted to see the most recent commit, you would run the command, *git log -1*.

--stat: This option shows some statistics on the number of changes (the number of inserted lines, deleted lines, and so on).

--pretty: This option allows you to specify format strings.

--format: This option allows you to create your own custom output format to see the different pieces of log output in nearly any format. (More on this in the section "Log Output Format.")

--oneline: This is a commonly used option when looking at history output. It tells Git to only display the first line of the commit message for each commit in the history. A best practice for working with Git is to form good commit messages, including making the first line meaningful. If the first line of the commit message is meaningful enough, it can be spotted easily among a larger set of output. (See Chapter 5 for more information on commit messages.)

--author: Another common filter is by author. Notice that the output of your log command shows a separate line for *Author:* that includes the username and the e-mail address associated with the commit (user.name and user.email as configured by git config). Anything in this *Author:* line can be searched for using this option. For example, if your name was *SCM User* and your e-mail address was scmuser@domain.com, then you could find commits with the *author* option using any of the commands below.

```
git log --author="SCM"
git log --author="User"
git log --author="domain.com"
```

--decorate: One other option that can be useful is *--decorate*. Whenever you see an option for *decorate*, this is telling Git to show references (names) that point to particular SHA1 values that represent particular commits. The most common references you'll see will be branch or tag names. So, the easiest way to think of this option is that it's telling Git to show any branch or tag names associated with the commit. To see the full namespace that Git uses to describe the reference, add the full option: --decorate=full.

Time-Limiting Options

Git allows for a number of time-based relative options such as --since, --until, --after, and --before. Two example forms of these options would be --since=2.weeks and --before=3.19.2015.

Notice that in these options, the dot (.) is used as a separation character between multiple parts. In certain cases, more freeform text is also allowed, as in these examples: --since "5 minutes ago" or --until "5 minutes ago".

These options can also be combined to create a bounded range:

```
git log --since 01.01.2015 --until 12.31.2015  or
git log --before 01.01.2016 --after 12.31.2014
```

History by Files and Paths

For the log command, you can provide filenames to the command line to filter the result. For example: `git log build.gradle` will show only the log entries where the file build.gradle was involved.

Adding the *--name-only* option to the log command will show the list of files changed with each commit along with the change information. So, using the *--oneline* and *--name-only* options is a convenient way to see the changes followed by the list of files, as in the following example.

```
$ git log --oneline --name-only
7ae0228 adding some initial testing examples
api/src/test/java/com/demo/pipeline/registry/V1_registryTest.java
build.gradle
4758f9e fix conflicts
f64bd2c update for latest changes to make web server generic and compatible with
tomcat
api/build/reports/tests/js/report.js
api/src/main/java/com/demo/pipeline/registry/V1_registry.java
...
b2e575a sql to recreate db
agents.sql
c6b5cbd update for db.properties
.gitignore
```

Adding a path onto the command (separated from the rest of the command by a double dash [--]) allows you to see just the commits that involved changes on that path.

```
$ git log --name-only -- web/src/main/webapp
commit f64bd2ce520e9a3df4259152a139d259a763bc31
Author: Brent Laster <bcl@nclasters.org>
Date:   Sun Mar 6 10:21:28 2016 -0500

    update for latest changes to make web server generic and compatible with tomcat

web/src/main/webapp/js/agents.js
web/src/main/webapp/js/delete.js

commit be42303ffb9356b8e27804ce3762afdeea624c64
Author: Brent Laster <bcl@nclasters.org>
Date:   Fri Feb 12 15:02:15 2016 -0500

    initial add of files from gradle build

web/src/main/webapp/WEB-INF/web.xml
```

```
web/src/main/webapp/agents.css
web/src/main/webapp/agents.html
```

Log Output Format

So what do you actually see when you do a Git log? Git shows you the series of commits that have occurred in (or been merged into) the current branch. An example is shown here using the *--oneline* option to simplify the output.

```
$ git log --oneline
482365d add exp function
d003b91 add max function
32d7b92 initial version
```

Notice that in the first column, you have the abbreviated SHA1 value. This is the SHA1 value associated with that commit. This is also your *handle* to that commit if you want to reference that commit in any future operation. The SHA1 in this case can be used like a version or revision number in other systems. Finding the unique SHA1 associated with a particular commit is a starting point for many workflow operations.

Git also provides ways to create a custom format to arrange and display fields in a particular style and order. Let's look at a more complex example:

```
$ git log --pretty=format:"%h ( %ad %an ) %s %d" --graph --date=short
```

This will produce output similar to the following example, which has multiple branches that have been created, modified, and merged back in to master. Although I haven't discussed branches in detail yet, you can get an idea from the indentation and colored lines and asterisks of the relationship between the branches.

The *--graph* option is basically a way for Git to show the relationships between branches (where new branches were created and work was done on them, where branches got merged back in, and so on) by using indentation and ASCII characters such as the slash and backslash.

```
*   65ad94b ( 2016-05-22 Brent Laster ) finalizing merge of branch1  (HEAD ->
master)
|\
| * 9f2d6c0 ( 2016-05-22 Brent Laster ) last update on branch1  (branch1)
* |   bc3be47 ( 2016-05-22 Brent Laster ) Merge branch 'branch2' into master
|\ \
| * | 28ac8bd ( 2016-05-22 Brent Laster ) last change on branch2  (branch2)
| * | ae90e52 ( 2016-05-22 Brent Laster ) add new file on branch2
* | | c9db77a ( 2016-05-22 Brent Laster ) update on branch1.5  (branch1.5)
| |/
|/|
* | da8862c ( 2016-05-22 Brent Laster ) update on branch1
|/
* 3b35284 ( 2016-05-22 Brent Laster ) update 2 on master
```

In this example, the other options have the following meanings:

--pretty defines the output format

%h is the abbreviated hash of the commit

%ad is the commit date

%an is the name of the author

%s is the commit message

%d specifies to show commit decorations (for example, branch identifiers or tags)

It is even possible to get the displayed columns to display in different colors if you want. And, as the text implies, *--date=short* tells Git to display date information in a short form.

As you can see, you can make log output display in almost any format you want. Using options such as *--oneline* and *--pretty=format* makes it easy to create output that is more easily consumable by another tool or process that needs to parse and read the Git history data.

If you often need to specify a long log command, trying to remember it all and retype it can be challenging. Fortunately, you can use another Git feature to simplify things: aliases (which I discussed in Chapter 4). Suppose you want to regularly run the git log command shown in the previous example because the output format is in a form you like. Instead of having to type out the command every time, you can create an alias for it. To do this, you would use the git config function.

```
$ git config --global alias.hist git log --pretty=format:"%h ( %ad %an ) %s %d"
--graph --date=short
```

By doing this, you have defined an alias named *hist* for this command and option string. From here on, you can just type `git hist` and you'll get the same output as if you were typing the full command string.

Searching History

Two other options available with the log command facilitate searching for text in files. These are often referred to as the Git *pickaxe*.

The first option, *-G*, takes a regular expression as an argument and searches for commits that added or removed occurrences of this text.

The second option, *-S*, takes a string and searches for commits that changed the number of occurrences of the string.

There are two differences here:

- -G is intended to take a regular expression, while -S normally takes a string. (You can tell Git to interpret the string for -S as a regular expression if you add the *--pickaxe-regex* option.)
- -S only detects situations where the before and after versions of a file have different counts of occurrences of the string. It won't flag an instance where a string was moved within a file, because that doesn't change the count of occurrences of the string.

As a quick example, suppose you have a file containing four lines in this order: **line1, line2, line3, line4**.

Now you add and commit that version. Next, you make a change in the file to switch the order of two lines so that the lines look like: **line1, line4, line3, line2**.

You add and commit that version, so now, your history has two versions: the first one with the lines in order and the second with the two lines swapped.

```
$ git log

commit 05fd71c23163487e9fa7d2fbb1580ab8b068f593
Author: Brent Laster <bcl@nclasters.org>
Date:   Tue May 24 14:38:12 2016 -0400

    reorder lines

commit a48586827b0737972b201bb0073f166f03c36bfe
Author: Brent Laster <bcl@nclasters.org>
Date:   Tue May 24 14:37:00 2016 -0400

    add file
```

If you now run the log -G command against these commits searching for ones containing *line2*, you'll see the following output:

```
$ git log -Gline2
commit 05fd71c23163487e9fa7d2fbb1580ab8b068f593
Author: Brent Laster <bcl@nclasters.org>
Date:   Tue May 24 14:38:12 2016 -0400

    reorder lines

commit a48586827b0737972b201bb0073f166f03c36bfe
Author: Brent Laster <bcl@nclasters.org>
Date:   Tue May 24 14:37:00 2016 -0400

    add file
```

If you run the log -S command against these commits searching for ones containing *line2*, you'll just see the following:

```
$ git log -Sline2
commit a48586827b0737972b201bb0073f166f03c36bfe
Author: Brent Laster <bcl@nclasters.org>
Date:   Tue May 24 14:37:00 2016 -0400

    add file
```

The reason for the difference is that the -G option detects the reordered lines as additions or removals, while the -S option is looking for a change in the number of occurrences. For the first revision, that is the first time those lines are present, so -S flags it. For the second revision, the lines have only been swapped—there is still the same number of occurrences of each line (1), so -S doesn't flag that one.

GIT BLAME

So far, you've learned how to view the history at a commit-by-commit level. For individual files, you can go a step further and see the revision that last modified each line. The command you use for this is git blame, which annotates the specified lines with information about the revision that changed it. The format is shown here.

```
git blame [-c] [-b] [-l] [--root] [-t] [-f] [-n] [-s] [-e] [-p] [-w]
[--incremental]
           [-L <range>] [-S <revs-file>] [-M] [-C] [-C] [-C] [--since=<date>]
           [--progress] [--abbrev=<n>] [<rev> | --contents <file> | --reverse
<rev>]
           [--] <file>
```

The main option to be aware of here is *-L* (lines). This option allows you to specify a range of lines to annotate if desired, instead of the entire file. It can take a start and stop value in the form of line numbers, a relative number of lines, or a regular expression.

Here are some examples from a Gradle-build script.

```
git blame build.gradle
4a4fe0ec (Diyuser          2016-02-14 22:39:36 -0500   1)
4a4fe0ec (Diyuser          2016-02-14 22:39:36 -0500   2)
1e8173ea (Brent Laster     2016-02-20 10:08:25 -0500   3)
bfb9b8de (Brent Laster     2016-04-08 00:21:04 -0400   4)
bfb9b8de (Brent Laster     2016-04-08 00:21:04 -0400   5) version =
'1.0.0-SNAPSHOT'
bfb9b8de (Brent Laster     2016-04-08 00:21:04 -0400   6) group = 'com.demo
.pipeline'
```

If you wanted to limit this to the lines between 5 and 10 (inclusive), you could use either `git blame -L5,10 build.gradle` or `git blame -L5,+6 build.gradle.` Both of these commands would return the following:

```
bfb9b8de (Brent Laster 2016-04-08 00:21:04 -0400  5) version = '1.0.0-SNAPSHOT'
bfb9b8de (Brent Laster 2016-04-08 00:21:04 -0400  6) group = 'com.demo.pipeline'
845bf97c (Brent Laster 2016-02-15 23:11:22 -0500  7)
4a4fe0ec (Diyuser      2016-02-14 22:39:36 -0500  8)
4a4fe0ec (Diyuser      2016-02-14 22:39:36 -0500  9)
^be42303 (Brent Laster 2016-02-12 15:02:15 -0500 10)
```

Notice the caret (^) sign at the beginning of the commit's SHA1 on the last line. This means that this is the *boundary commit*, the earliest commit in the range of commits being examined. If you prefer, you can pass the *-b* option to show a blank field for boundary commits instead.

You can also use regular expressions for the start and stop of the line ranges. To do that, you specify the expression inside the forward slashes (/ /). For example, to look at seven lines starting after the occurrence of the string *subprojects*, you would use the following:

```
 $ git blame -L"/subproject/",+7 build.gradle
^be42303 (Brent Laster 2016-02-12 15:02:15 -0500 11) subprojects {
^be42303 (Brent Laster 2016-02-12 15:02:15 -0500 12)    apply plugin: 'java'
^be42303 (Brent Laster 2016-02-12 15:02:15 -0500 13)    apply plugin: 'eclipse-wtp'
1e8173ea (Brent Laster 2016-02-20 10:08:25 -0500 14)    apply plugin: 'jacoco'
```

```
^be42303 (Brent Laster 2016-02-12 15:02:15 -0500 15)    version = '1.0.0-SNAPSHOT'
^be42303 (Brent Laster 2016-02-12 15:02:15 -0500 16)    group = 'com.demo.pipeline'
^be42303 (Brent Laster 2016-02-12 15:02:15 -0500 17)    configurations.compile
.transitive = true  // Make sure transitive project dependencies are resolved.
```

The ^ symbol appears on all lines that were changed before the first commit in the range—in this case before the current commit.

You could also use regular expressions to see the blame annotations for entire blocks by passing in regular expressions that signify the syntax for the beginning and end. An example of doing this for your build script to see the *dependencies* closure would be as follows:

```
$ git blame -L"/dependencies {/","/}/" build.gradle
^be42303 (Brent Laster      2016-02-12 15:02:15 -0500 53)       dependencies {
^be42303 (Brent Laster      2016-02-12 15:02:15 -0500 54)               compile
'mysql:mysql-connector-java:5.1.38'
00000000 (Not Committed Yet 2016-05-21 10:21:19 -0400 55)       //      compile
'mysql:mysql-connector-java-bin:5.1.38'
^be42303 (Brent Laster      2016-02-12 15:02:15 -0500 56)               compile
"javax.ws.rs:jsr311-api:1.1.1"
```

Notice the use of the regular expression "/dependencies {/" as the starting value and the regular expression "/}/" as the ending value to bound the area of interest.

You can also specify a range of revisions to limit the git blame output. You can pass these revisions to the command without needing a separate option. If you want to just use the latest revision as an end point of your range, you can use the double dash (--) syntax to specify that.

So, if you were to use the git log command to see the list of revisions for a file, you could then use the blame command and pass a revision to see who has made changes since that revision.

```
$ git log --oneline  build.gradle
7ae0228 adding some initial testing examples
f64bd2c update for latest changes to make web server generic and compatible with
tomcat
be42303 initial add of files from gradle build

$ git blame f64bd2c .. -- build.gradle
...
^f64bd2c (Brent Laster 2016-03-06 10:21:28 -0500  75)
^f64bd2c (Brent Laster 2016-03-06 10:21:28 -0500  76) project(':dataaccess') {
7ae02289 (Brent Laster 2016-03-29 14:27:13 -0400  77)
...
7ae02289 (Brent Laster 2016-03-29 14:27:13 -0400  84)   }
7ae02289 (Brent Laster 2016-03-29 14:27:13 -0400  85)   task testJar(type: Jar) {
7ae02289 (Brent Laster 2016-03-29 14:27:13 -0400  86)             classifier "test"
```

When specifying a starting revision, if a change was made earlier than that revision, it will still show up in the output as *blamed* on that starting revision. Because this is like a boundary revision (at the bounds of what you specified), you can hide that SHA1 with the -b option if desired. Another way to see only what has been changed since the starting revision would be to use a simple grep to filter out the boundary revision.

```
$ git blame f64bd2c.. -- build.gradle | grep -v "\^"
```

Another useful option allows you to specify revisions as relative time ranges. The typical syntax is *<number of units>.<type of units>* as in *5.days* or *2.weeks*. However, you can also supply dates in the form yyyy-mm-dd. To specify these date-and-time values, you can use the same time options as on the log command: *--since, --before, --until,* and *--after*. Here are some examples.

```
$ git blame --after=2016-03-28 build.gradle
$ git blame --since=2.weeks build.gradle
$ git blame --before="8 weeks ago" build.gradle
```

The date specified here defines a boundary, and so any change in the file that happened before that date gets *blamed* on the revision closest to that change. This can make parsing the output challenging, because all the lines from before that timestamp still show. To work around this, you can pipe the blame output through the grep command I mentioned previously. Here's an example:

```
$ git blame --after=2016-03-28 build.gradle | grep -v "\^"
```

Finally, there is a *--reverse* option that reverses the timeline searching. This option displays the last commit where a line existed before it was deleted. To use this, you pick a commit where you know the line existed and then run the command searching in reverse from that commit to HEAD.

```
$ git blame --reverse <SHA1 of commit in past where you know line existed>..HEAD --
<filename>
```

Git blame can also follow changes across file renames, copies, and so on. For more information on that functionality, see the git blame documentation.

SEEING HISTORY VISUALLY

Most instances of Git come packaged with a utility named *gitk*. It is invoked by running gitk in a terminal session and allows you to browse the history of your local repository in a graphical interface. Note that this is looking at the history in the local repository and *not* in the remote repository. Figure 7-1 shows what the gitk interface looks like.

In the top-left pane is a listing of changes made in the repository. The branching structure is represented in a graph layout, similar to how the --graph option formats the command line output. In the top-right pane are details about the commits on the left. You can also select commits and see more details about them in the lower panes of the interface. For example, notice the SHA1 values that can be selected to look at parent commits.

Near the center of the interface are two radio buttons—Patch and Tree—that specify how you select items to navigate around the changes displayed in gitk. If you are new to the tool, it is not always intuitive to navigate around the history via the Patch mode. Until you get comfortable with this mode, you should select the radio button for Tree. This will display a directory and file layout similar to a typical explorer interface that may be easier to navigate.

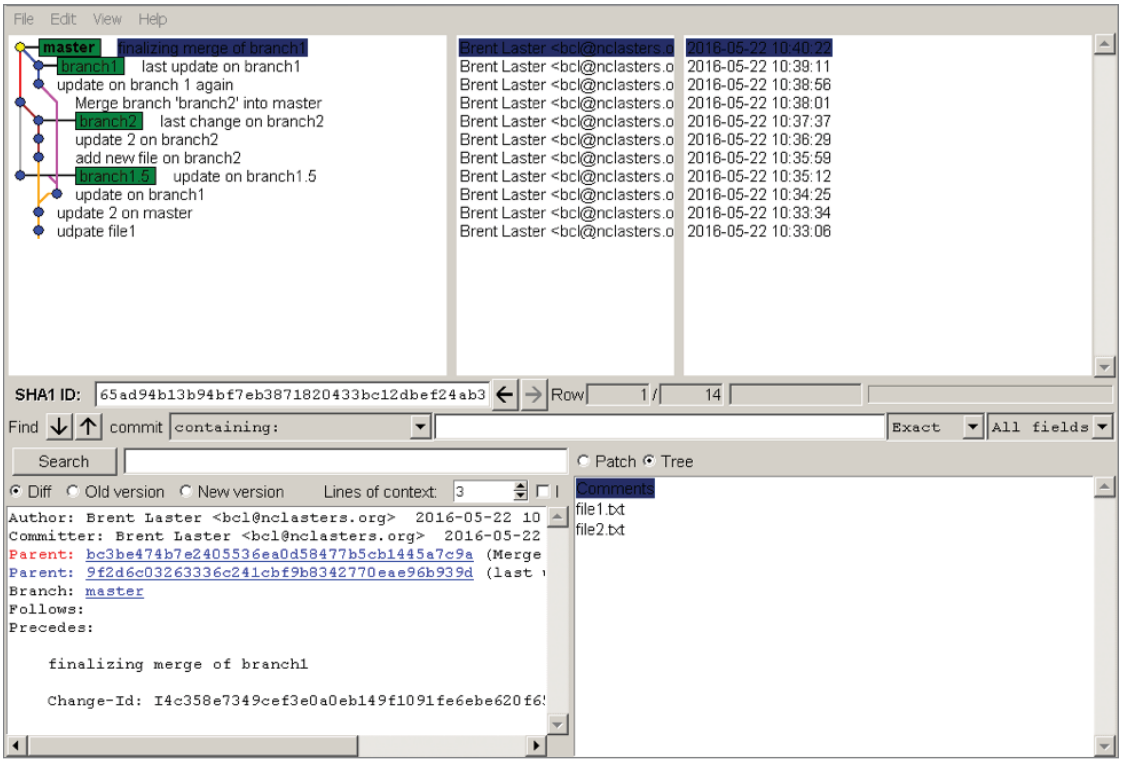


FIGURE 7-1: Using the gitk tool to browse local history

TAGS

When working with specific commits in Git, it is useful to have a symbolic name as an alias for the commit's SHA1. In Git, as in most source management systems, you do this with *tags*. A tag is just a symbolic name you attach to a specific commit. For example, if you wanted to refer to a commit as *RelCandidate1*, you could use the following command: `git tag RelCandidate1 <SHA1 value>`.

Afterward, that tag would be associated with that commit and serve as a persistent alias to Git to reference that commit. Figure 7-2 shows an example.

Notice that no matter how many more commits are made, the tag stays with this commit. An important point here is that you are tagging snapshots and commits, not individual files.

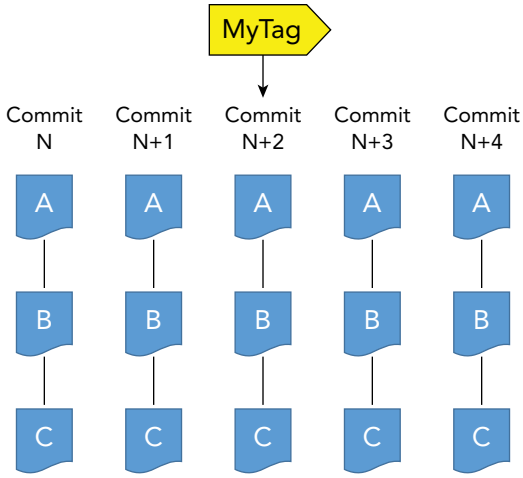


FIGURE 7-2: Tagging a commit

In Git, this kind of tag is called a *simple* tag. Git also provides a second kind of tag known as an *annotated* tag. An annotated tag in Git also includes a record about who created it, when, and a commit message to describe it. In short, it is like committing a change to the repository. To create this kind of tag, you would use the command `git tag -a <rtag> <SHA1 value> -m "message"`.

There is one more important point regarding tags in Git. As I mentioned, you are tagging snapshots, not individual files. This means that if you apply an operation to a tag, that operation will be against all files associated with that tag, just as it would be for any SHA1.

So, if you have 20 files in your commit snapshot, and you label that commit as *RC1*, then this command `git log RC1` will list the history for all 20 files that are part of the snapshot. If you only want to see the history for one file, you need to supply that filename as an additional qualifier on the command, as in `git log RC1 <filename>`.

Seeing Tag Details

If you want to see the list of tags associated with a repository, you can just run the tag command without any options. The command would be `git tag`.

If you just want to see details for a tag, you can use the show command and pass the tag name, as in `git show <tag>`.

If the tag is a simple tag, then you will just see the details of the commit record. If the tag is an annotated tag, then you will see the metadata associated with the tag as well.

If you just want to verify the SHA1 of the commit associated with the tag, you can use one of Git's plumbing commands, *rev-parse*, in the format `git rev-parse <tag>`. This will return the SHA1 value that is tagged with that name.

Modifying Tags

Normally, if you try to change what a tag points to, you will get an error from Git. Suppose that you have an existing tag, *tag1*, that you have attached to revision 5128459. If you then try to update that tag to point to another revision, you'll see this:

```
$ git tag tag1 4e430fe
fatal: tag 'tag1' already exists
```

To work around this, you need to supply the -f option.

```
$ git tag -f tag1 4e430fe
Updated tag 'tag1' (was 5128459)
```

To delete a tag, you can use the -d option.

```
$ git tag -d tag1
Deleted tag 'tag1' (was 4e430fe)
```

The last modification that's useful to know how to do is renaming a tag. For this, you use a slight trick. First, you create a new tag that points to the old tag; then you delete the old tag.

```
$ git tag oldtag 4e430fe
$ git log --oneline --decorate | grep 4e430fe
4e430fe (tag: oldtag) update 2 on branch2


$ git tag newtag oldtag
$ git log --oneline --decorate | grep 4e430fe
4e430fe (tag: oldtag, tag: newtag) update 2 on branch2

$ git tag -d oldtag
Deleted tag 'oldtag' (was 4e430fe)
$ git log --oneline --decorate | grep 4e430fe
4e430fe (tag: newtag) update 2 on branch2
```

WARNING *When modifying a tag, if content with that tag has already been pushed to the remote repository, then there are additional considerations. In particular, unless you push the changes over to the remote repository explicitly, they won't show up there. I cover how to do that in Chapters 12 and 13.*

Additionally, if others may already be using the code from the remote repository with the old tag, they will need to be made aware of any changes you plan to do in a coordinated way so they can get the updated tag. In general, if a tag already exists in code on the remote side, the simplest approach is just to add a new tag and leave the old one, unless there is a specific reason you need to have the old one removed.

Quick Tagging Example

Suppose that you have the following commit history in your local repository:

```
$ git log --oneline
65ad94b finalizing merge of branch1
9f2d6c0 last update on branch1
bc3be47 Merge branch 'branch2' into master
28ac8bd last change on branch2
ae90e52 add new file on branch2
c9db77a update on branch1.5
```

If you want to create an annotated tag against ae90e52, you can use something like the following:

```
$ git tag -a annotatedTag1 ae90e52 -m "Creating an annotated tag"
```

If you then do a log command and add the --decorate option, you can see the tag in the history.

```
$ git log --oneline --decorate
65ad94b (HEAD -> master) finalizing merge of branch1
9f2d6c0 (branch1) last update on branch1
bc3be47 Merge branch 'branch2' into master
28ac8bd (branch2) last change on branch2
ae90e52 (tag: annotatedTag1) add new file on branch2
c9db77a (branch1.5) update on branch1.5
```

To see the details of the tag, you can use the show command. (Note that this will also show the thing pointed to by the tag—the commit—but will show the annotated tag information first.)

```
$ git show annotatedTag1
tag annotatedTag1
Tagger: Brent Laster <bcl@nclasters.org>
Date:   Sun May 22 21:22:54 2016 -0400

Creating an annotated tag

commit ae90e529cf198f5d53a2654f08d69f791f8b6d88
Author: Brent Laster <bcl@nclasters.org>
Date:   Sun May 22 10:35:59 2016 -0400

    add new file on branch2

    Change-Id: I8614ae7233c934e03c652c85c21894146ef362b5

diff --git a/demo/file2.txt b/demo/file2.txt
new file mode 100644
index 0000000..ef49dd8
--- /dev/null
+++ b/demo/file2.txt
@@ -0,0 +1 @@
+more
```

UNDOING CHANGES IN HISTORY

At some point, most Git users will want a way to undo changes and get back to a previous state in the local repository, staging area, working directory, or all three. Git provides two ways of accomplishing this through the *reset* and *revert* commands. Both commands allow you to get to the desired set of content, but they take different approaches. And depending on which one you use, you can have a non-trivial impact on other users.

Reset—Rolling Back Changes

You can think of the purpose of the reset command as performing a *rollback* function, undoing a set of changes, and getting back to a previous state. Effectively, the reset command moves the HEAD of your local repository back to a previous commit, and, optionally, updates the staging area and working directory with the contents of that previous commit.

The basic form of the reset command is as follows:

```
git reset [-q] [<tree-ish>] [--] <paths>...
git reset (--patch | -p) [<tree-ish>] [--] [<paths>...]
git reset [--soft | --mixed [-N] | --hard | --merge | --keep] [-q] [<commit>]
```

> **NOTE** *Occasionally, you will see the terms* commit-ish *and* tree-ish *in Git documentation. Though odd sounding, the names are fairly self-explanatory: something like a commit, and something like a tree.*
>
> *To be more specific, the* ish *part can be thought of as implying* something that can resolve to *the term that came before it. So, a commit-ish is something that can resolve to a commit, and a* tree-ish *is something that can resolve to a tree in Git's datastore. Internally, Git deals with multiple kinds of objects in a hierarchical fashion. There are four main ones: annotated tags that generally point to a commit; commit objects that point to the root of a structure; tree objects that represent directories in the structure and point to other trees or blobs; and blobs that are essentially files.*
>
> *Commands that take commit-ish objects expect something that either is a commit or can be dereferenced or followed to get to a commit. An example of the latter case would be a tag that can be dereferenced to a commit. Commands that take tree-ish objects expect something that either is a tree or can be dereferenced or followed to get to a tree. And, because all commits ultimately point to a tree object, all* commit-ish *objects are* tree-ish *by definition. However, the reverse is not true: not all tree-ish objects are commit-ish.*
>
> *In most cases, you can just think of the <tree-ish> and <commit-ish> items here as placeholders where you can plug in a value that equates to a SHA1.*

The first and second forms of the command populate the staging area with the specified revision. In the third form, the options represent which parts of the local environment should be updated to match the contents of the new SHA1. The options are as follows:

soft—only update the HEAD of the local repository.

mixed—(default) update the HEAD of the local repository and the staging area.

hard—update the HEAD of the local repository, the staging area, and the working directory.

Usually, to roll back to a previous commit, you need to first find the SHA1 of that commit. You can do that through the *git log* command. Using the --oneline option provides concise output that you can choose from. Then, to point back to that commit, you can use "`git reset <SHA1>`" where <SHA1> is the SHA1 value for the commit you want to reset to.

This moves the HEAD pointer in the local repository to point back to the commit indicated by the SHA1. Because no option was specified, and *mixed* is the default option, this also updates the staging area with the contents of the commit pointed to by the new HEAD.

> **NOTE** *When you specify a SHA1 value for the reset, you can use either an* absolute *SHA1, as I have been discussing, or a* relative *SHA1. Git has special syntax that allows users to specify SHA1s relative to another SHA1. The simplest example would be specifying something like* 1 before the current *or* back 1. *In Git, using the caret (^) symbol after a reference means* 1 before. *For example, because HEAD is the current commit on the current branch, to reset back to 1 before the current, you can run the command* `git reset HEAD^`.
>
> *This is sometimes called the* caret parent. *If you want to go further back relative to the current commit, you can use more carets or use the "~#" syntax. To get back to 3 before the current commit, and only update the local repository, you can use either* `git reset --soft HEAD^^^` *or* `git reset --soft HEAD~3`.

Completely Resetting the Local Environment

If you want to completely reset your local environment back to the most current commit in the local repository, wiping out any uncommitted changes you had, you can use the --hard option with HEAD as in `git reset --hard HEAD`.

In this case, HEAD is already at the current commit, so there's no change there. What does change (because you're using the --hard option) is that Git updates the staging area and the working directory with the contents of that commit—overwriting anything in those areas and effectively resetting them to the current content of the repository. One way to remember this is that if you hit your *head* against something *hard*, it hurts. So, if you use the hard option and then realize you did need something that was just overwritten, that's likely to *hurt*.

One note here for clarity: when you use the reset command, you are only moving pointers (references) in the repository as far as rolling back—you are not deleting any existing commits.

Revert—Canceling Out Changes

While reset is useful for rolling back to a certain point, it can also be problematic. The difficulty occurs if a reset is done locally on content that has previously been pushed to the remote repository. If you are changing things in that code base that was on the remote, there is a chance that you will impact other users who have cloned down the latest changes and are working against those pushed changes.

In that scenario, if a reset is done and changes are made, and then pushed to the remote, there can be some interesting merge challenges when the next person goes to merge in their content, based on where HEAD used to point.

For these reasons, it is recommended to not use reset or any Git operations that change history and could cause difficult merge scenarios on code that has been pushed to a remote repository.

So how do you do a rollback if you can't use reset in a particular situation because it's too problematic? One way is to check out an old version of the content and then check it in on top of the current version. However, Git provides a more automated, elegant solution: *git revert*.

What *revert* does is try to cancel out the effects of changes made in commits you don't want. For example, suppose you recently committed a change that added two lines and you want to get back to the commit before that one. The `git reset HEAD^` command moves HEAD back in the repository to point to the previous commit. As noted, this can open up the possibility of some merge issues if the code has been previously pushed out to the remote repository and others are working with it.

You can, however, use *git revert*. In this case, `git revert HEAD^` examines the different commits and creates a new commit to cancel out the changes—that is, it creates a new commit that deletes those two lines.

The difference here is that revert is adding content that cancels out the effects of the change as opposed to pointing back to the old content. Adding content at the end of the branch does not cause the same kind of merge issues you might encounter from a reset. Dealing with additional changes is a common occurrence when working with remote repositories. Changing history or rolling back to previous commits on the remote side is not.

> **NOTE** *Revert has some behavior and options that you should be aware of.*
>
> *First, when you issue a revert command, by default, it opens the editor to allow you to type in a commit message, because it plans to do a commit to cancel out the changes. To suppress this, you can use the --no-edit option. Do not use the -m option. The -m option in this case has to do with a special case of reverting a merge.*
>
> *There is also a --no-commit (-n) option. This allows you to revert things, but only in the working directory and staging area. The user still has to do a separate commit operation to make the change in the local repository.*

Figures 7-3 - 7-7 show examples of some git reset and revert commands. The setup here in Figure 7-3 is that you have your local environment with the local repository, staging area, and working directory. There have been three commits of content, with each commit changing the file to add a line. HEAD points to the latest commit, which is also tagged as *current*.

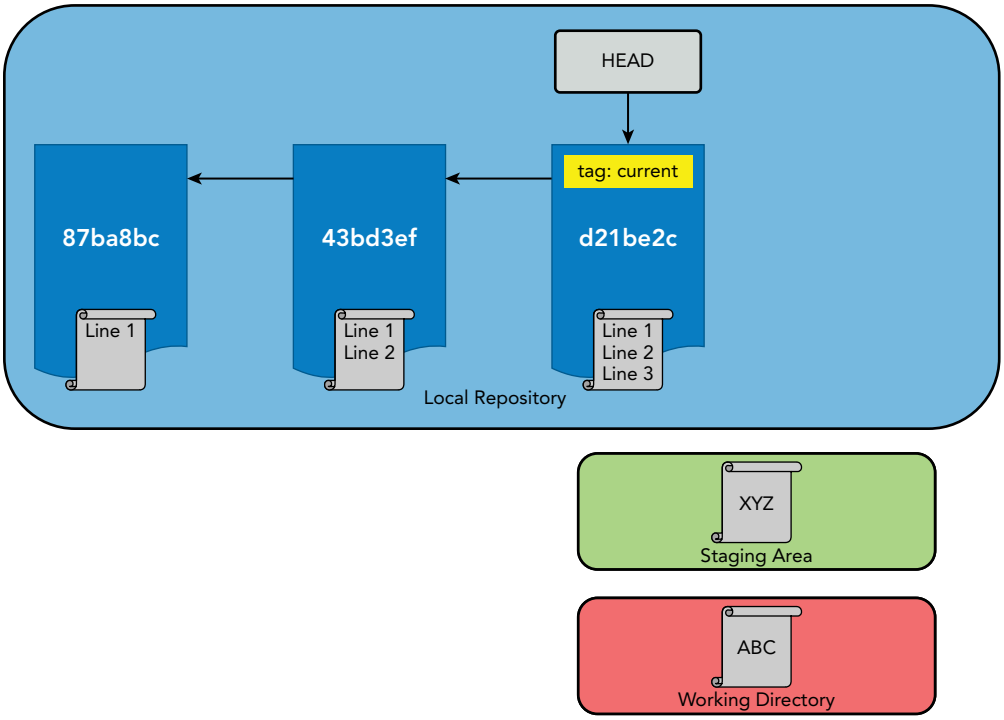


FIGURE 7-3: Starting repository contents

In Figure 7-4, you are doing an *absolute* reset to an exact SHA1 that is two commits before the current one. You are also using the hard option, which updates both the staging area and the working directory from the contents of the new HEAD.

In Figure 7-5, you are resetting relative to the tag -1 before. (Note the syntax of ~1 meaning *1 before*.) This effectively moves HEAD back 1 commit *and* updates the staging area with the contents of that commit because *mixed* is the default.

Now, in Figure 7-6, you start back at the original spot and then issue a revert command.

After the revert in Figure 7-7, you have a new revision added that effectively cancels out the effect of the commit you want to ignore.

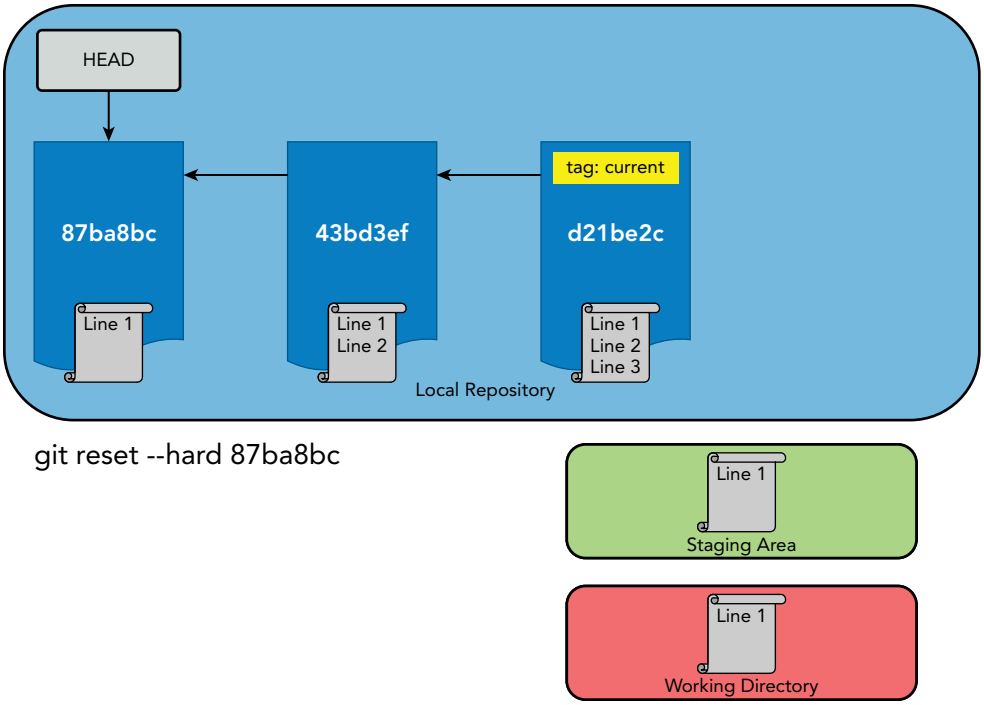


FIGURE 7-4: Resetting back to an absolute SHA1

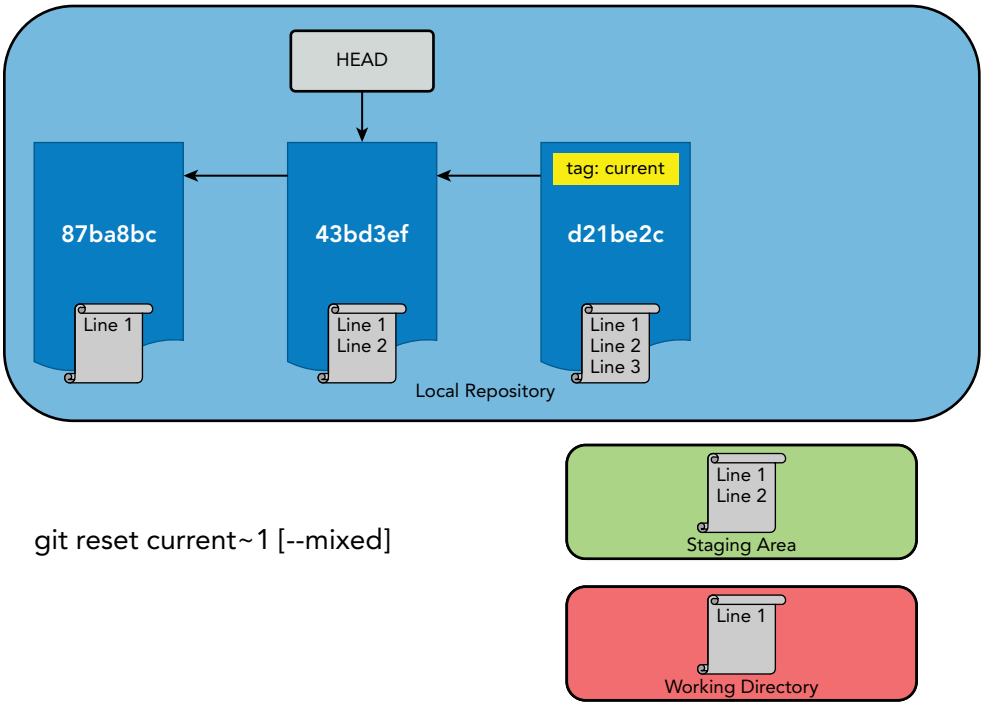


FIGURE 7-5: Resetting relative to a tag

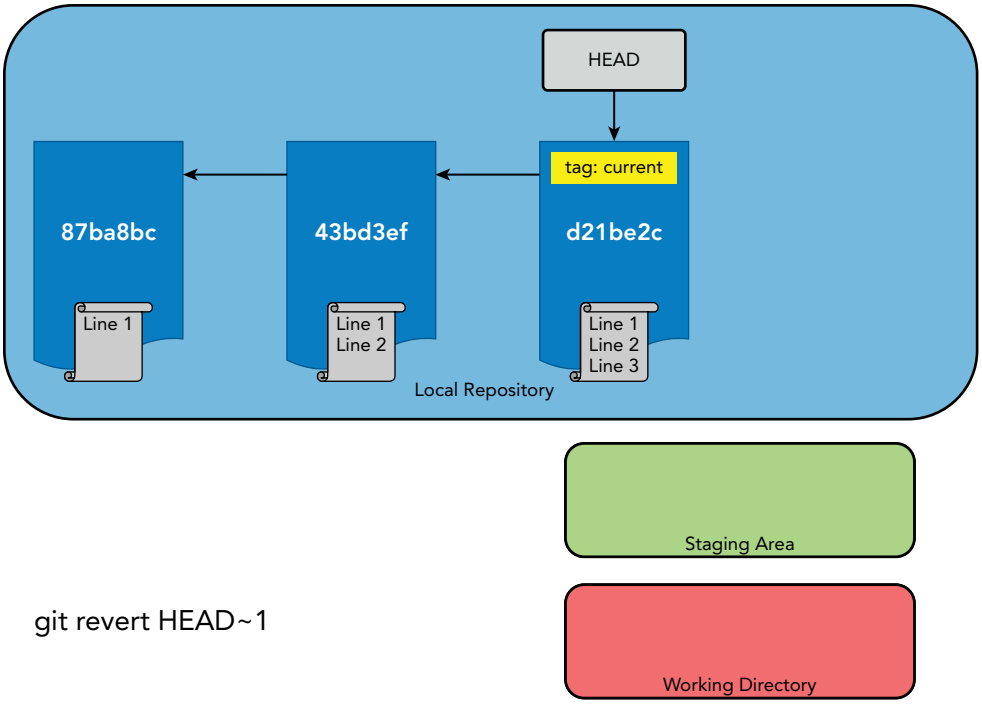


FIGURE 7-6: Resetting for revert

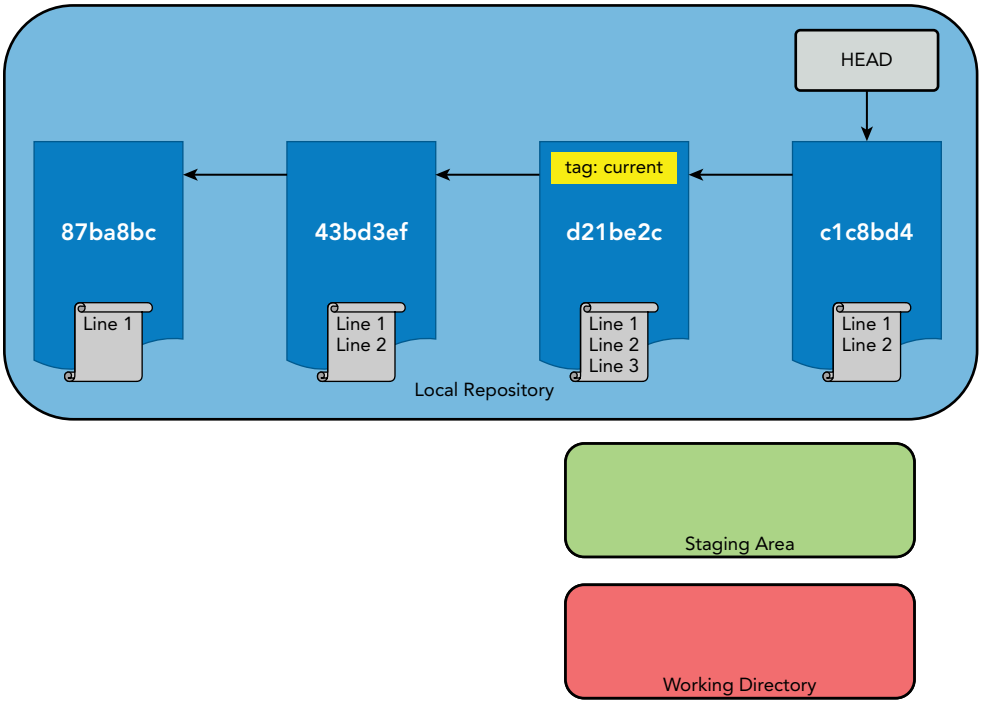


FIGURE 7-7: Local environment after the revert

ADVANCED TOPICS

In this section, you look at adding additional security to tags and commits by signing. You also look at another kind of log, called a reflog, that Git uses to track changes to references over time.

Signing Commits and Tags

As I have mentioned, Git creates SHA1 values for everything it stores internally. Aside from serving as unique identifiers for the items in Git's content-addressable store, these SHA1 values ensure that nothing can be modified without Git knowing about it.

This represents security within Git. But what if you want to guarantee that commits you are incorporating from someone else are actually from them? Or, what if you want a way to signify that something coming from you or a member of your team is actually from you or them? The user.name and user.email properties can be set to anything by any user before making a commit. For added security, users can *sign* commits or tags in Git using the GNU Privacy Guard, better known by its acronym, *GPG*.

GPG can be downloaded from `http://www.gnupg.org`. Once you download GPG and set it up, you need to set up a new key—unless you already have a key installed.

To see what keys (if any) you have installed, you can use `gpg --list-keys`. You see something like the following:

```
/Users/dev/.gnupg/pubring.gpg
-----------------------------
pub   2048D/00D026C4 2010-08-19 [expires: 2018-08-19]
uid       [ultimate] GPGTools Team <team@gpgtools.org>
uid       [ultimate] GPGMail Project Team (Official OpenPGP Key) <gpgmail-devel@
lists.gpgmail.org>
uid       [ultimate] GPGTools Project Team (Official OpenPGP Key) <gpgtools-org@
lists.gpgtools.org>
uid       [ultimate] [jpeg image of size 5871]
sub   2048g/DBCBE671 2010-08-19 [expires: 2018-08-19]
sub   4096R/0D9E43F5 2014-04-08 [expires: 2024-01-02]

pub   4096R/DBB83F35 2016-05-21 [expires: 2020-05-21]
```

If you need to create a new key, you can use `gpg --gen-key`. This will prompt you for several settings. For most of the prompts, you can just accept the suggested default values (for the prompts that have defaults). GPG asks you to enter your name, your e-mail, and a comment, and then to verify it. You are then asked to enter and verify an optional passphrase.

After completing that process, you have a key that looks similar to the previous example, something like this:

```
pub   2048R/B34AA7EA 2016-05-21
      Key fingerprint = B8E5 4910 5D1D 7655 AA08  88B8 933B 9040 B34A A7EA
uid       [ultimate] B. C. Laster (demo signing key) <bl2@nclasters.org>
sub   2048R/59ABEFED 2016-05-21
```

Now you can tell Git that you want to use that key to sign things. You do this by setting a configuration value, `user.signingkey`, to the value after the slash from the pub section. Afterward, this key is available in Git to sign tags and commits.

```
$ git config --global user.signingkey B34AA7EA
```

Signing Commits

Once the key is set up, signing a commit just requires adding the -S option to the commit command. Here's an example:

```
$ git commit -S -am "update"

You need a passphrase to unlock the secret key for
user: "B. C. Laster (demo signing key) <bl2@nclasters.org>"
2048-bit RSA key, ID B34AA7EA, created 2016-05-21

[foo 4173266] update
 1 file changed, 1 insertion(+)
```

Git log includes options to see which commits have valid signatures and to see the actual signatures. Using formatted output, you can add the G option, as shown here:

```
$ git log --pretty=format:"%G? %h %ad | %s%d"
G 4173266 Sat May 21 07:20:19 2016 -0400 | update (HEAD -> foo)
G 54c01fa Sat May 21 06:55:11 2016 -0400 | signing commit
N 4400ff6 Thu Jan 28 21:57:30 2016 -0500 | update (master)
N d8e53c4 Mon Jan 25 23:17:07 2016 -0500 | lower.c
```

From this example, you can see that the last two commits are signed (by the presence of the G).

To see more details, you can use the --show-signature option.

```
$ git log --show-signature -1
commit 41732668cd9d49166e007ce64fe4d862fa366f99
gpg: Signature made Sat May 21 07:22:01 2016 EDT using RSA key ID B34AA7EA
gpg: Good signature from "B. C. Laster (demo signing key) <bl2@nclasters.org>"
[ultimate]
Author: local user <bl2@nclasters.org>
Date:   Sat May 21 07:20:19 2016 -0400

    Update
```

Signing Tags

Creating a signed tag is similar to creating an annotated tag, except that the -a option is replaced by the -s option.

```
$ git tag -s my_tag -m "creating a signed tag"

You need a passphrase to unlock the secret key for
user: "B. C. Laster (demo signing key) <bl2@nclasters.org>"
2048-bit RSA key, ID B34AA7EA, created 2016-05-21
```

Running `git show` on the tag also now shows the signature.

```
$ git show new_tag
tag new_tag
```

```
Tagger: local user <bl2@nclasters.org>
Date:   Sat May 21 08:00:19 2016 -0400

adding a signed tag
-----BEGIN PGP SIGNATURE-----
Comment: GPGTools - https://gpgtools.org

iQEcBAABCgAGBQJXQE3aAAoJEJM7kECzSqfqtjcH/2PV+RIRQChJ7in5A0lvj4KB
X5+Onk3lKKZMT/1wGRo8bvsyafSKkxwlFYogLZun+fsf4dQgB8e6LTBLOVnegHB9
3SNHtLxJ/C+lnVLjZWQ1fuiW9iiWZOovSrwVRz4yM3yqKLVxgxhJE6ol72O1gjjT
AMQiBZccKABOqPZAnFwehMZ4Rv8pcnmIkZ6Ost3CV2Sp2wvNWks5VVBDzjpjU84G
VNpHnL+VEOtybe22P/QoYsndUu7HruPx5Q5bxPNcE7aAPTlka4bmkL2CgCCBHwnl
h0x9WzSzEFZuynQBO7iN6pdHh1bOlufK97gLgkib56JSnu1hv9NOH690b0Og0/Y=
=0bbD
-----END PGP SIGNATURE-----
```

Reflogs

I'll briefly touch on one other type of log here: reference logs (or *reflogs*). A *reference* in Git terminology refers to a name that you use to refer to a SHA1 value. For example, the default branch in Git is master, and, at any point when using Git, master refers to the current content on the branch. This is the SHA1 of the most recent commit. You could reference that commit by its SHA1 value or by referencing *master*.

In the Git repository in the .git directory, references are stored in a path structure. At the top is *refs*, then the type, and then the name. For example, the current SHA1 for the current commit in the master branch would be stored in *.git/refs/heads/master*. If you are on the master branch, then this should match up with the SHA1 from the latest change.

```
$ cat .git/refs/heads/master
373f47835befd4bc24f5b0109eb96a305a15863e
$ git log -1
commit 373f47835befd4bc24f5b0109eb96a305a15863e
Author: Brent Laster <bl2@nclasters.org>
Date:   Tue Mar 29 20:39:20 2016 -0400
remove extraneous files
```

Now, as content is committed into the repository, new SHA1s become the most current and the SHA1 values in the reference files change. Other things, such as switching branches, can cause other references to change, such as HEAD, which tracks the current branch.

While there is only one current value for any of these references, a reflog for each reference records the values as they change over time. Being able to see how these values change provides another record of what has been done in the system. Also, as you'll see in Chapters 8 and 9, the reflogs can provide useful information on past points you may want to go back to in the history of the reference. They also record the points where branches are changed.

The *git reflog* command has several different options, including options to prune entries from the logs. Those options are not typically used by Git users. Instead, the most common way to use them is with the default option to *show* the log entries. Some examples follow:

```
$ git reflog
150d863 HEAD@{0}: commit: saving Docker image files
```

```
bfb9b8d HEAD@{1}: commit: copies of latest work
373f478 HEAD@{2}: checkout: moving from master to new2
...
845bf97 HEAD@{9}: commit: update script
4a4fe0e HEAD@{10}: commit: updated to publish to artifactory
b2e575a HEAD@{11}: clone: from http://github.com/brentlaster/roarv2

$ git reflog master
373f478 master@{0}: pull origin new: Fast-forward
1e8173e master@{1}: commit: add in sample tests and jacoco configuration
...
4a4fe0e master@{5}: commit: updated to publish to artifactory
b2e575a master@{6}: clone: from http://github.com/brentlaster/roarv2

$diyuser@diyvb:~/roarv2$ git reflog new2
150d863 new2@{0}: commit: saving Docker image files
bfb9b8d new2@{1}: commit: copies of latest work
373f478 new2@{2}: branch: Created from HEAD
diyuser@diyvb:~/roarv2$ git reflog new
373f478 new@{0}: pull origin new:new: storing head
```

You can then use these relative values shown in the reflog as points to reset to as in:

```
$ git reset --hard HEAD@{3}
```

SUMMARY

In this chapter, you learned how to use the git log command to create formatted history output. You were introduced to the gitk tool, which allows you to see history on your local repository in a graphical presentation.

You learned about the tag functionality in Git that allows you to tag revisions—both simple and annotated. You also learned about two ways to effectively roll back changes in Git—the reset command and the revert command—and when to use (or not use) each one.

In the Advanced Topics section, you learned how to sign commits and tags for additional verification and security. Finally, you looked at reflogs, another form of log that Git keeps as references change over time.

In the next chapter, I'll introduce branches and start explaining how they work and the power they provide you in Git.

About Connected Lab 4: Using Git History, Tags, and Aliases

To better understand these concepts, you can do the Connected Lab following this chapter. It will allow you to practice the concepts presented here as you work with the log command and create aliases and tags.

Connected Lab 4

Using Git History, Aliases, and Tags

In this lab, you'll work through some simple examples of using the git log command to see the flexibility it offers, and also create an alias to help simplify using it. You'll also look at how to tag commits to have another way to reference them.

PREREQUISITES

This lab assumes that you have done Connected Lab 3: Tracking Content through the File Status Life Cycle. You should start out in the same directory as that lab.

STEPS

1. Starting in the same directory that you used for Connected Lab 3, begin by making another change to the repository to make the history more interesting. Add a line to the first file you committed into the repository and then stage and commit. Note that you can use the following shortcut:

```
$ echo new >> file1.c
$ git commit -am "add a line"
```

2. Look at the history you have so far in your small repository. To do this, run the log command. (In some terminals, your history may be longer than the screen and so you will need to press a key to continue. If you are paging through the log output on a Unix terminal and want to end the listing, press the q key.)

```
$ git log
$ git status -s
```

3. Often when looking at Git history information, users only want to see the first line of each entry, the *subject line*. This is why it is important to make that first line meaningful when using Git. (Note that I do not do that in this book.)

   To see only the first line of each log message, you can use the --oneline option. Try it now.

```
$ git log --oneline
```

4. You can create a more complex version of the log command that includes selected pieces of history information formatted in a specific way. (Refer to Chapter 7 or the log command help page to clarify what each part of this command is doing.) Be careful of your typing: note the colon after *format*, the double hyphens, and the double quotes.

```
$ git log --pretty=format:"%h %ad | %s%d [%an]" --date=short
```

5. Because this is a lot to type, you can create an alias to simplify running this command. You do this by configuring the alias name to stand for the command and its options. Type the following, paying attention to the punctuation (double hyphens, colon, vertical bars, single and double quotes, and so on).

```
$ git config --global alias.hist 'log --pretty=format:"%h %ad | %s%d [%an]"
--date=short'
```

6. Run your new hist alias. You will see the same output as the original log command from step 4. If you encounter any problems, go back and double-check what you typed in step 5.

```
$ git hist
```

7. You can also use the log command (and your hist alias) on individual files. Pick one of your files and run the hist alias against it.

```
$ git hist <filename>
```

8. You're interested in seeing the differences between a couple of the revisions. However, there are no version numbers. So we'll need to use a different way with Git. In Git, we specify revisions using the SHA1 (hash) values (the first 7 bytes are enough). In our hist output, the first column is the SHA1 value.

9. Run the git hist alias again and find the SHA1 values of the earliest and latest lines in the history. (Yours will, of course, be different from mine in the following example.)

```
       $ git hist

  latest -> 1db49cf 2016-08-20 | add a line (HEAD -> master) [Brent Laster]
            ece66a5 2016-08-20 | committing another change [Brent Laster]
            8103190 2016-08-20 | update [Brent Laster]
            581c751 2016-08-20 | another update [Brent Laster]
earliest -> c6a82d2 2016-08-20 | first commit [Brent Laster]
```

10. You can use these SHA1 values similarly to how you might use version numbers in other systems. Take a look at the history between your earliest and latest commits. To do this, you run the hist alias and specify the range of values using the SHA1 values. Execute the following command, substituting the appropriate SHA1 values from the history in your repository. (Use the format, git diff <earliest SHA1>..<latest SHA1>.)

```
$ git hist c6a82d2..1db49cf
```

11. You see a similar history to what you saw previously. One thing to note here is that you don't see the original (first) commit. This is because when specifying ranges using the ".." syntax, Git defines that syntax as essentially everything after the first revision. (See the section on "Specifying a Range of Commits" in Chapter 9 for more information.) Note that you can also run this command against an individual file. Try the following command with your SHA1 values and the first file you added in the repository. (Use the format, git hist <earliest SHA1>..<latest SHA1> <first file>.)

```
$ git diff c6a82d2..1db49cf file1.c
```

12. This is useful, but finding and typing SHA1 values each time for operations like this can be cumbersome. Instead, you can use tags to point to commits, and then use those tag names instead of the SHA1 values in commands. You'll now create tags for the earliest and latest commits in your repository, using the tags *first* and *last*, respectively. The commands are as follows (using the format, git tag <tagname> <hash>):

```
$ git tag first c6a82d2
$ git tag last 1db49cf
```

13. Now that you have the tags, you can use them anywhere you used the SHA1 values before. Try out the hist alias with the tags.

```
$ git hist first..last
```

14. You may not have thought about it, but this is giving you the history for all of the files in the repository. This is because a tag applies to an entire commit, not a specific file in the commit. To see this more clearly, add the --name-only option to the command and run it again.

```
$ git hist first..last --name-only
```

15. If you only want to do an operation using a tag for one file, you can simply add the filename onto the command, as in the following example:

```
$ git hist first..last --name-only file1.c
```

8

Working with Local Branches

WHAT'S IN THIS CHAPTER?

- Understanding branches
- Implementing branches in Git
- Branching operations
- Branching workflows and models
- Performing checkouts on non-branched commits
- Working in detached HEAD mode
- Performing checkouts on individual files

In this chapter, I introduce Git's fast, lightweight, and yet very powerful branching model. I clarify the different kinds of pointers and symbolic names that come into play, show what happens when you create, modify, and change branches, and explain how Git allows users to easily work with multiple branches.

Git actually manages two categories of branches: remote branches (updated in the remote repository) and local branches (updated in the local repository). For the purposes of this chapter, I'm only talking about local branches, and I'll just use the term *branch*. I discuss remote branches in Chapter 12.

WHAT IS A BRANCH?

In a traditional source management system, there are multiple ways of describing what a branch is. From a usage standpoint, a branch is usually thought of as a *line of development*, meaning a set of code allocated for working on a release or update for a product. From an

implementation standpoint, it is usually a collection of specific versions of a group of files that are labeled and accessed with a common identifier.

Example from Another Source Management System

To dive into this idea a bit more, let's look at an example from the CVS world. This is only for illustration purposes; you don't have to know CVS. In CVS, you can create a branch with a command like the following:

```
cvs rtag -a -D <date/time> -r DERIVED_FROM  -b NEW_BRANCH PATHS_TO_BRANCH
```

The pieces of this command translate in the following ways:

- **cvs rtag.** Create a tag in the repository for all of the indicated content. The *rtag* here refers to a repository tag, that is, a branch name.
- **-D <date/time>.** Tag these files based on a specific date and time. You want to associate the branch with versions of these files as they are (and were) at a certain point in time.
- **-r DERIVED_FROM.** The other branch you are using as the *parent* for this branch.
- **-b NEW_BRANCH.** The name of the branch to create, and thus the name of the repository tag.
- **PATHS_TO_BRANCH.** The set of files and directories to associate with the new branch.

So what's the end result of this command? A user can now reference and work with specific versions of all of the files in the repository paths (independent of other versions) by using one unique identifier, the branch name.

Continuing the CVS example, if you want to get the versions of all of the files associated with the branch name down to your working area, you can do a checkout with *cvs co -r BRANCH_NAME PATHS*.

What you end up with in your working area is a set of specific versions of files, or what you could also refer to as a … *snapshot*. That term should sound familiar.

The Git Model of Branches

So how does this all relate to Git? Well, think about what a snapshot is in Git. Here are two ways you might describe it:

- A line of development, associated with a specific change.
- A collection of specific versions of a group of files associated with a specific commit.

The end result when you create a commit is that you have a *handle* to get all of the versions of the files in the repository associated with that commit—the commit's SHA1 value. When you later retrieve that snapshot, what you end up with is a versioned set of files in your working area—the version identified and tied to that SHA1 value.

By now you may be thinking that a snapshot in Git sounds suspiciously like a branch, and you're right. In fact, every snapshot in Git has the potential to become a branch. All it needs is an identifier that has the branch name and points to it.

This means that, in Git, a branch can be created by simply adding a lightweight, movable pointer to a SHA1 value for a particular commit—simple! Let's look at how this is modeled in the system.

Take a look at Figure 8-1. Each snapshot that is put into a Git repository becomes a new commit. Each commit already has the contents of a potential branch within it.

By default, Git establishes a branch named *master* for commits. This is implemented by having a pointer named *master* that points to the current commit. As new commits are made, the master pointer advances to point to each new commit. In this way, when master is referenced, it always points to the latest commit in the repository, just as a branch in a traditional system will contain the latest checked-in code.

Each commit also has a pointer back to the commit that came before it. This builds up a chain, which is useful in situations such as determining the correct ancestors or the basis for a merge (discussed in Chapter 9).

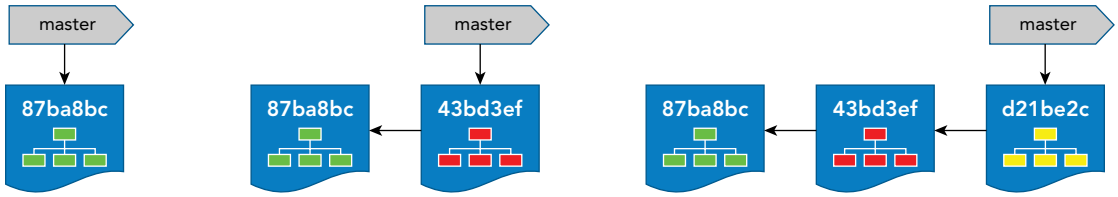


FIGURE 8-1: Progression of chain of commits

In a traditional SCM system, each file that is part of a branch is updated with a branch tag or identifier. To get and reference the collection of files that make up the branch, the system searches and collects all of the files with that identifier. In Git, the collection is already in the snapshot, so there's no tagging and branching multiple files and file trees. The collection of content is already directly referenced by using the SHA1 value, so a branch name can point to that SHA1 value and reference the same set of content.

At the implementation level, creating a branch in Git simply involves writing a 41-character file named for the branch. Within the file, the 41 characters are the SHA1 value of the commit that the branch points to (40 characters plus a newline). Because this is so simple, branches in Git are extremely quick and easy to create (and delete).

> **NOTE** *You may be wondering what the difference is between a tag and a branch in Git. Both are references (pointers), but a tag is attached to the commit it was originally created for, while a branch changes which commit it points to as content is added to the branch. In simplest terms, one is a stationary pointer (tag), while the other is a movable pointer (branch) to commits.*

Creating a Branch

The command to create a branch in Git is *git branch*.

```
git branch [--color[=<when>] | --no-color] [-r | -a]
      [--list] [-v [--abbrev=<length> | --no-abbrev]]
      [--column[=<options>] | --no-column]
      [(--merged | --no-merged | --contains) [<commit>]] [--sort=<key>]
      [--points-at <object>] [<pattern>...]
git branch [--set-upstream | --track | --no-track] [-l] [-f] <branchname>
[<start-point>]
git branch (--set-upstream-to=<upstream> | -u <upstream>) [<branchname>]
git branch --unset-upstream [<branchname>]
git branch (-m | -M) [<oldbranch>] <newbranch>
git branch (-d | -D) [-r] <branchname>...
git branch --edit-description [<branchname>]
```

In its simplest form, *git branch <branch name>* tells Git to create a branch called <branch name> starting with the contents of the current branch.

Figures 8-2 through 8-5 show what happens internally in Git when a new branch is created. Starting with Figure 8-2, you're at the point from above where master has had several commits made to it.

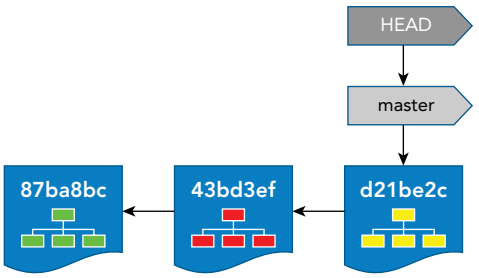


FIGURE 8-2: Your starting chain of commits

In this figure, you'll notice another pointer labeled *HEAD* pointing at master. You may recall the term HEAD from the discussion about diffing in Chapter 6. I suggested thinking of it as pointing to the current commit on the current branch. In fact, HEAD's primary job is to identify what is most current, and, in particular, which branch is currently active. In this case, because HEAD is pointing to master, this means that master is your current (active) branch. Commands that you issue will go against master (unless another branch is specified).

Now, you'll create a new branch named *testing*. The command to do this is *git branch testing*.

When you issue this command, the following events happen:

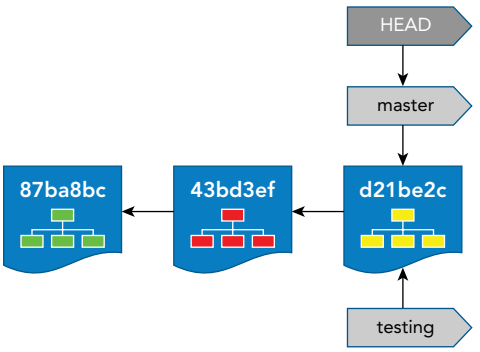


FIGURE 8-3: After the creation of a testing branch

- A new pointer named *testing* is created that points to the most current commit (the same one that master currently points to). (Illustrated in figure 8-3.)
- Within Git, a new reference file is created as */refs/heads/testing* (in the .git directory) and the SHA1 value of the current commit is recorded in it.

Checking Out a Branch

Once a branch exists, you can get content from it using the *checkout* command. The checkout command has the following syntax:

```
git checkout [-q] [-f] [-m] [<branch>]
git checkout [-q] [-f] [-m] --detach [<branch>]
git checkout [-q] [-f] [-m] [--detach] <commit>
git checkout [-q] [-f] [-m] [[-b|-B|--orphan] <new_branch>] [<start_point>]
git checkout [-f|--ours|--theirs|-m|--conflict=<style>] [<tree-ish>] [--] <paths>...
git checkout [-p|--patch] [<tree-ish>] [--] [<paths>...]
```

So, if you want to get the latest version from testing, you can use this checkout command: *git checkout testing*.

The checkout command causes Git to perform two actions:

1. Switch the active branch to be *testing*.
2. Check out the content (from the commit currently pointed to by *testing*) into the working directory.

It's worth noting here that when Git checks out content, in many cases it will overwrite the version in the working directory. You should ensure that local content is committed, stashed (discussed in Chapter 10), or otherwise saved before doing a checkout over the top of existing content.

To elaborate more on the first action, remember that Git keeps track of which branch is the *active* one by using the special pointer HEAD. At the time a checkout is done, if the branch being checked out is different from the current one, Git updates HEAD to point to the specified branch. Figure 8-4 illustrates the state after creating and checking out the *testing* branch.

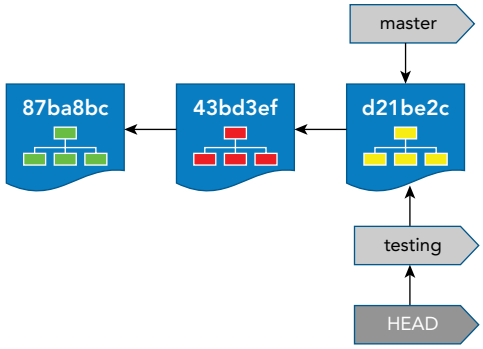


FIGURE 8-4: After checking out the testing branch

Subsequent commands then operate by default on the new active branch until another checkout switches branches (by updating HEAD) again.

> **NOTE** *In the preceding examples, I used two commands to create and then switch to (check out) the testing branch. The two commands were*
>
> 1. git branch testing
> 2. git checkout testing
>
> *Git provides a shortcut to perform these two operations via one command: the checkout command with the -b option. So, for the testing case, the syntax would be* `git checkout -b testing`.

At any point when using Git, there is always an active branch. By default, this is master until other branches are created and switched to by a checkout.

Adding Content to a Branch

When a new commit is made into the local repository, it becomes the next *link in the chain* to the current commit. The *current commit* here is the commit pointed to by the branch reference that HEAD points to. After the new commit is added to the repository, the pointer for that branch is moved to the new commit.

In other words, HEAD points to the current branch, and the current branch pointer is updated to point to the new commit. This has the net effect of *updating* the branch to a new version of code. Figure 8-5 illustrates this process.

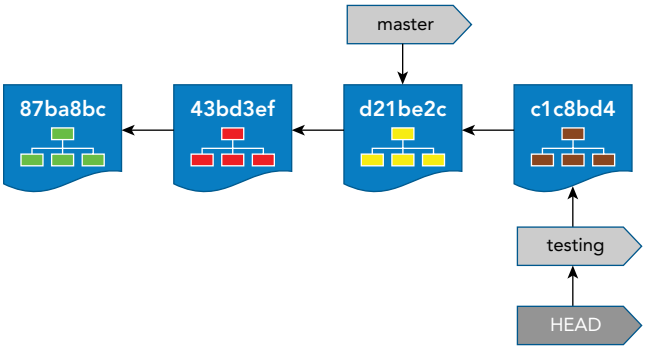


FIGURE 8-5: The current branch pointer is moved to indicate that the newest commit is the latest content on that branch.

As subsequent commits are made, the branch pointer for the same branch advances to the latest commit, until the branch is changed. Note that the other branch pointer (master) does not advance, because that branch is not the active one and so is not being updated.

One Working Directory—Many Branches

If you have previously worked with multiple branches in a source management system, you likely had multiple working areas—one for each branch—to avoid mixing versions of content. It was up to you to manage keeping all of the working areas up to date, and remembering which one was which.

Git simplifies this process by doing one additional, but highly useful, step when you switch branches with checkout. When you do a Git checkout, Git updates your local working directory with the *checked out* files from the commit the branch points to. If you then switch branches again, Git will update your working directory with the *checked out* versions of the files from the new branch.

Put another way, Git ensures that the content in your working directory is consistent with whatever branch you switch to. This is one of the key mind shifts of learning and using Git: you only have and need one working directory for a repository, no matter how many branches are in it. This takes some getting used to, but is ultimately a very useful feature. It is key to supporting workflow models such as creating content in separate branches and merging them back into integration or production branches.

> **NOTE** *A recent addition to Git, the worktree functionality, allows users to have multiple working directories associated with a single Git repository. This provides an alternate way of working with multiple branches – separating them out into different working directories. This is not the typical way of using Git, but can provide advantages for certain use cases. Worktrees are discussed in detail in Chapter 14.*

So, when you check out and switch branches, Git causes several things to happen:

- It moves the HEAD pointer to point to the branch you are switching to.
- It updates the content in your working directory to the latest *flat files* from the branch you are switching to.
- It updates the indicators that tell you which branch is the active one.

Let's take a look at how this works in practice. Figure 8-6 represents two parts of your local environment: the local repository and the working directory. (I've omitted the staging area for simplicity.) In the local repository, you are at the same place that you left off after creating the testing branch in the previous section. Note that HEAD currently points to master, so that is your current branch.

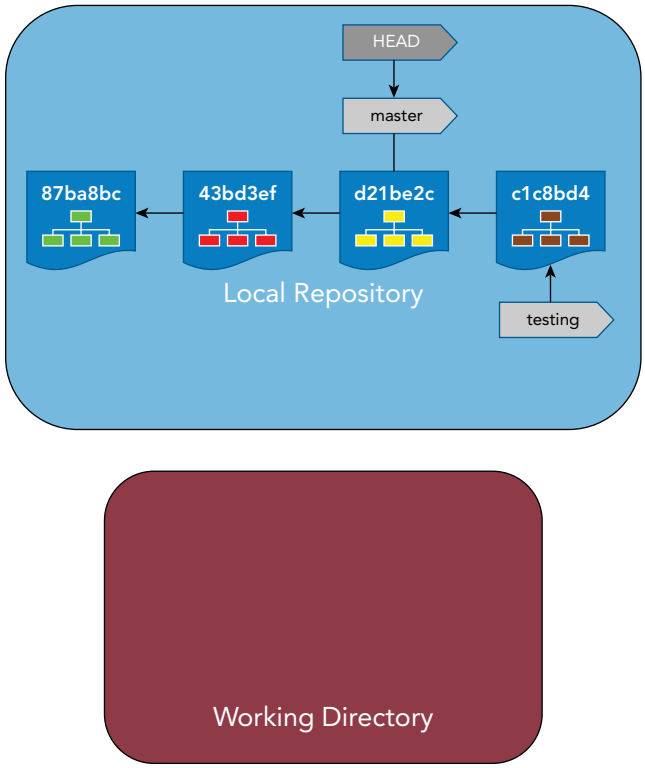


FIGURE 8-6: Local repository—active branch: master

Now, assume you issue a checkout command. Because master is the active branch, Git checks out the current commit pointed to by master and updates the content in the working directory. Figure 8-7 illustrates this.

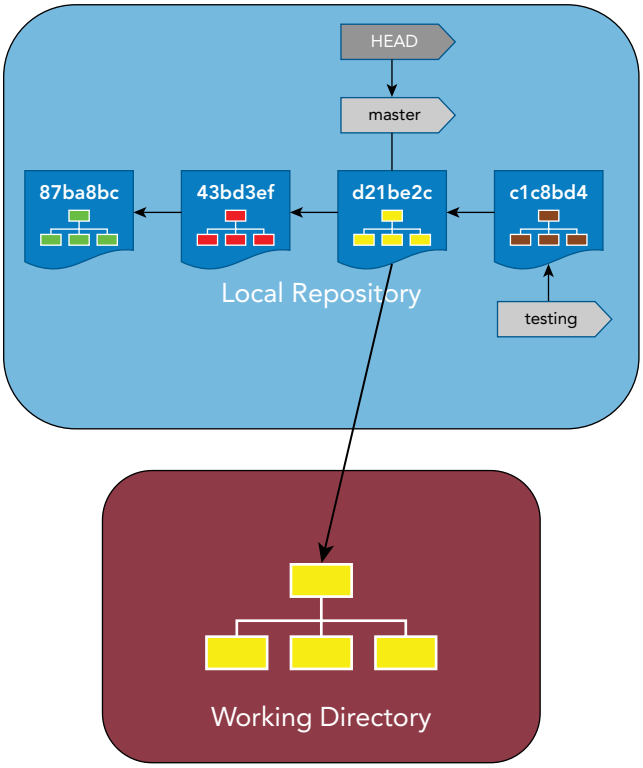


FIGURE 8-7: Git checkout master

Next, you decide you want to do some work in the branch named *testing*. To switch to that branch, you can just do another *git checkout*. Figure 8-8 illustrates this.

Notice what happens when you run this command. The HEAD pointer switches to point to the branch pointer *testing*, making testing the active branch. Then Git checks out the content in the commit pointed to by the testing branch into the working directory—the same place it checked out the master branch into previously!

Git took care of updating the content in the working directory to be from the branch that you told it you wanted to work with—with no effort on your part other than to run the checkout command.

This means that you can work with as many branches as you want *in the same repository* and only use one working directory. Git ensures the content in the working directory is from the correct branch.

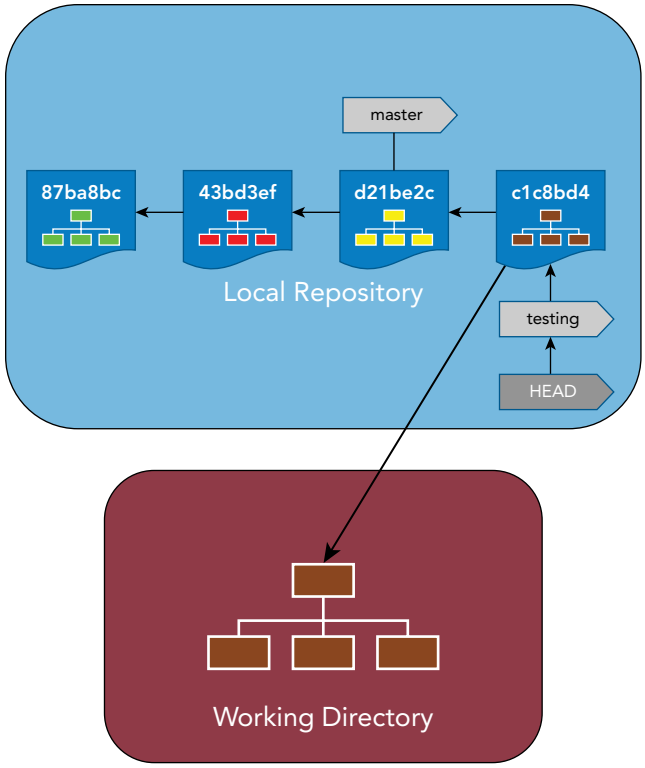


FIGURE 8-8: Git checkout testing

Just to prove the point, if you were to check out master again, you would see Git switch back to master and update the working directory with the contents of the latest commit associated with master. Figure 8-9 illustrates this state again.

There is one caveat that can come into play when trying to check out (switch to) another branch in Git. If you have work in progress (tracked files with uncommitted changes) in your working directory, you have to move these files out of the way before Git will allow the switch.

Simply put, if Git detects that you might lose work by overwriting uncommitted changes with content from the other branch, it will refuse to do the checkout. It wants the working directory to be clean in the sense of having all content tracked by Git up-to-date with Git (your latest versions committed). If you have this situation, it can be handled in one of three ways:

- Commit any uncommitted changes; *or*
- Stash any uncommitted changes using Git's stash command (I talk about how this works in Chapter 11); *or*
- Use the --force option to go ahead and do the switch, overwriting any uncommitted changes.

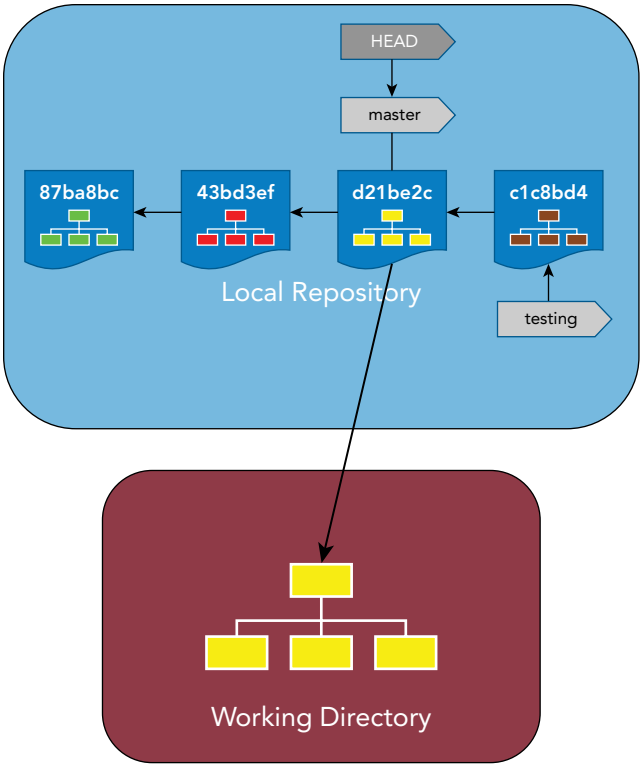


FIGURE 8-9: Git checkout master (again)

In case it's not obvious, the last option is not recommended because it can result in work being overwritten.

Getting Information about Branches

I've already mentioned that Git maintains an internal pointer, HEAD, to keep track of which branch is current and active. But how do I as a user know which branch is current?

If you're working on the Git command line, you can simply run the command *git branch*. In this simple form, the command lists the local branches and places an asterisk (*) by the one that is currently active.

```
$ git branch
  master
* testing
```

In many terminal sessions, your prompt will also be configured to show the branch, as in

```
mymachine (master) $
```

NOTE *If you don't already have the Git branch as part of your prompt and want to set it, you can do this yourself via some configuration in one of the dot (.) files on a Unix system. Here's an example of code to do this in a .profile.*

```
__git_prompt
{
    local b="$(git symbolic-ref HEAD 2>/dev/null)";
    if [ -n "$b" ]; then
        printf " (%s)" "${b##refs/heads/}";
    fi
}
```

For bash, you need to enable bash-completion *and then you can use this command (as an example):*

```
export PS1='$(__git_ps1) \w\$ '
```

Alternatively, there are more elaborate scripts on websites such as GitHub, and plenty of references on how to do this on the web.

Another useful option for quickly viewing information about your branches is the *-v* or *--verbose* option. This option provides a list of the branches and a quick one-line summary for each of the last commits on the branch.

```
$ git branch -v
  master  05fd71c reorder lines
* testing 8250f9b update for testing
```

Finally, when you have a lot of branches and need to locate one or more of them quickly, there is the *--list* option. This option lists the local branches, but more importantly, it can take a pattern as an option and filter the list by that pattern. Here are some examples:

```
$ git branch --list
  master
* testing

$ git branch --list t*
* testing

$ git branch --list *ter
  master
```

Deleting or Renaming a Branch

The *branch* command includes options to allow you to delete or rename a branch. Because they are similar in syntax and usage, I'll cover them both here.

Deleting a Branch

To delete a branch, you can use the -d option, as in *git branch -d <branch name>*. In some cases, when performing a delete, you might see a warning message like this from Git:

```
$ git branch -d test
error: The branch 'test' is not fully merged.
If you are sure you want to delete it, run 'git branch -D test'.
```

In this case, Git is telling you that you have commits on the branch that you are about to delete that may not be reachable through any other branches if you proceed. In effect, you would *lose* being able to reference them from a branch.

For example, consider the branch arrangement from the checkout example earlier (see Figure 8-10).

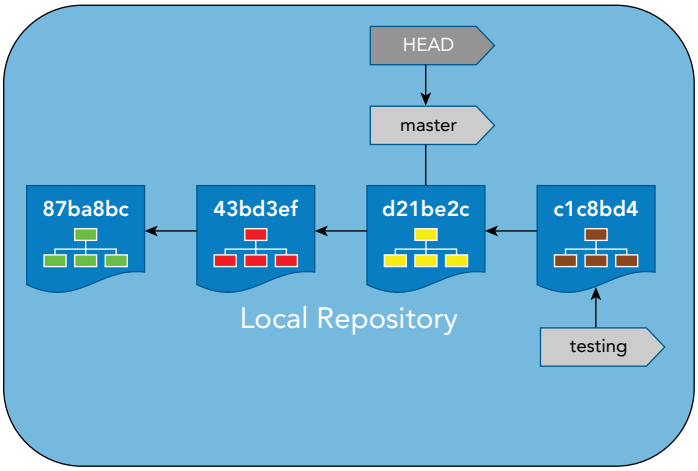


FIGURE 8-10: Local repository with two branches

If you were to delete the *testing* branch at this point, you would lose that pointer to commit *c1c8bd4* (see Figure 8-11).

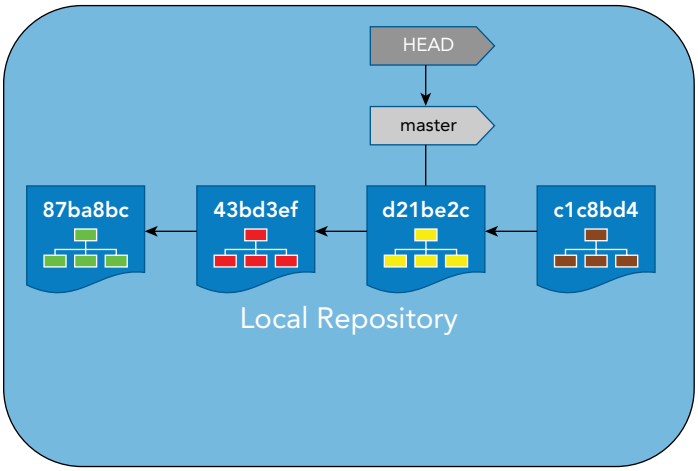


FIGURE 8-11: After deleting the testing branch

> **NOTE** *In reality, it's very hard to actually ever* lose *anything in Git. Notice that even though you got rid of the branch pointer as a way to access the commit in the previous example, the commit is still there.*
>
> *In fact, as long as you know the SHA1 value of the commit, you can still connect to it, via something like a tag,*
>
> ```
> $ git tag <tagname> c1c8bd4
> ```
>
> *or even create a new branch based on it,*
>
> ```
> $ git branch <branchname> c1c8bd4
> ```
>
> *At some point, if an object in Git is truly orphaned and doesn't have any connections, and git gc (garbage collection) is run, then objects can be deleted. However, this usually requires an explicit user-based action to execute.*

If you aren't sure of which commits might be lost if you continue, you can use a variant of the *log* command to find out. Here is the format:

```
$ git log delete_candidate_branch ^current_branch --no-merges
```

The caret (^) symbol at the start of the second branch argument means *not*. So the way to read this command is *Show me the commits that are in delete_candidate_branch and are NOT in current_branch*. The --no-merges option tells Git not to include commits that have been merged in already.

Translating that to the example, you would run the command,

```
$ git log testing ^master --no-merges
```

> **NOTE** *You may be thinking that you've seen the caret (^) used with a revision before and it meant something different. You're right. In Chapter 7, I talked about using* reset *and* revert *with revisions specified as HEAD^.*
>
> *In Git, when the caret sign is at the end of a reference, it means* 1 back from, *so in this case it would be* 1 back from HEAD.
>
> *When the caret sign is at the front of a reference, it means* not - *as in* not in this chain.

After doing the evaluation, if you decide you want to proceed with the deletion, you can force it to happen in one of two ways: `git branch -d -f testing` or `git`.

Here, -f is short for --force and, as the name implies, it overrides the warning message and executes the operation. -D is an alias for *-d -f*.

Renaming a Branch

To rename a branch, you can use the -m option, as in `git branch -m <current name> <new name>`. Here, again, if you run into conflict situations (such as the <new name> already existing), Git stops the operation and warns you.

```
$ git branch -m test testing
fatal: A branch named 'testing' already exists.
```

If you are sure you want to continue and execute the operation, you can use similar options as you have for deleting a branch: `git branch -m -f test testing` or `git branch -M test testing`.

Here, -f is short for --force and, as the name implies, it overrides the warning message and executes the operation. -M is an alias for -m -f.

Developing with Branches

In this section, I'll build on the branching fundamentals and extend them to common concepts working within the Git model. I'll discuss some ideas of where to do development, as well as models for integrating work back into a production area.

Using Branches for Initial Development

As I discussed in the previous sections, Git has a powerful branching model. You've seen how its internal implementation makes branches very easy to create and use. This is in stark contrast to how most users and admins think about branching in traditional source control systems. In those systems, as the scope of the repositories grows, so does the apprehension and complexity associated with branching—particularly when dealing with merge conflicts. Because of these concerns, users may steer away from branching, even when it would be beneficial. They may also abdicate responsibility and oversight for branching to specialized persons or support groups.

The Git world flips this model on its head—emphasizing the use of branches as cheap and easy—with operations that are simple for any user to perform. The workflow when using Git is intended to be branch-based. In this model, separate branches can be (and are) created for almost any development activity, whether it's trying out something new, creating a new feature, or implementing a bug fix. Then, when the work in the individual branch is ready, it is merged into a production branch. In some cases, there may also be one or more integration branches where smaller changes are merged together and verified before they are merged into the production branch. This process repeats throughout development iterations.

I'll now talk about some different ways to use branches, which are also called *branch classifications*.

A REMINDER ABOUT REPOSITORY SCOPE

A quick reminder before I talk more about the branching workflow model in Git: choosing the right scope for a repository can simplify things significantly. I've mentioned these considerations in earlier chapters but they are worth repeating here.

Branching is a quick operation because it is just creating a reference to a commit with the branch name. However, checkout performance and space are dependent on the number of files and directories in the repository. In most cases, overhead will never be noticed, but it is something to keep in mind if you are working with a very large repository.

And while I'm on the subject of large repositories, even a repository that has a small number of files and directories can still be large if it has a lot of large files stored in it—especially large binaries that are hundreds of megabytes. In general, Git can do a good job of compressing similar content in the repository, or *packing*. However, large binaries don't fit as well into that model.

You can tell Git that a file is binary through an entry in the Git attributes file (discussed in Chapter 10) or, even better, you can store and track your large binary files in an artifact repository such as Artifactory so they are not part of your Git repository.

Aside from the physical makeup of your repository, another important consideration is the amount of content covered by the files in the repository. The broader the content, the more likely that multiple users will be making changes in the repository, and so merging becomes a bottleneck as everyone tries to keep up with each other's changes. Again, think of the scope of repositories when you are setting them up or migrating from another system.

If the scope resembles the scope of a development project in an IDE or other interface (for example, a jar in Java), the chances of being impacted by others and having to do merges are significantly less than if you had the entire product in one repository.

Ways to Use Branches (Classifications)

As you've seen, creating branches in Git is quick and easy. In fact, creating, changing, merging, and deleting branches in Git can be as quick as working with files in some other systems.

Given this speed and ease of use, it's common practice when working with Git to create a branch to work on anything new before merging it into the main production area. Along these lines, you can broadly classify branches based on their intended use. I cover some of the related classifications of branches in the following sections.

Topic Branches

When you are working on something in Git that is under development and not yet in production, you should create a branch. Then you can switch to that branch and develop, test, and so on in a separate area. In the end, you may decide that this line of development is not useful. That's fine; you can delete the branch if you end up not using it, or merge it back in when appropriate if you do use it.

Branches created for short-term development efforts are frequently referred to as *topic branches*, the idea being that branching in Git is so lightweight you can create a branch for any topic. Ultimately, though, topic branches are intended to be short-lived, as they are for initial development use until content is merged into a production or integration branch.

A topic branch is most typically used by one user on a local system, unless there is a need to collaborate in the content with other users. Keeping it local helps reduce the amount of *clutter* in the repository. A convention that is sometimes used to identify one of these branches is to create a name for the type of function or operation being explored and prefix it with a *namespace* (similar to a directory path) consisting of the initials of the user. This can be specified easily at the time the branch is created. Here's an example (assuming initials abc): `git branch abc/web_client_port`.

One other point about these kinds of branches: because they are generally experimental or exploratory in nature, they are usually not subject to the same kind of formal processes as other branches. Policies on what can be done in this kind of branch are usually more relaxed. For example, they may not go through the continuous delivery cycle or require issue tracking or approvals for pushing. However, best practices would include some kind of quality control or gates before a topic branch was merged into another, more *formal* branch.

Feature Branches

A feature branch (as the name implies) is intended for developing a feature of a software product or something on a similar scope—typically larger than the scope of a topic branch, for example. This may include work by several developers where the work comes together on the remote side when pushed.

A feature branch is still a temporary branch with a limited lifetime. However, the intention from the beginning is that this will be merged into the production code when complete. For that reason, feature branches are treated more formally, with more oversight. They usually participate in many of the same processes as production branches, such as utilizing issue tracking and control for approving pushes. There is also a review when a feature branch is ready to be merged into a higher-level branch.

Integration Branches

Another category of branch that can be utilized in workflows is an *integration branch*. An integration branch may be a formality or a necessity, depending on the number of developers involved, the number of features being worked on, and the complexity of the overall project.

As the name implies, integration branches serve as integration points between feature development and merges into the production or release branch. They are a place to integrate work together and make sure things work as expected, before merging the larger set of changes into the branch for releasing code. Basically, they serve as a buffer to help prevent destabilizing the code targeted for production.

These kinds of branches mandate a higher level of monitoring, oversight, testing, and so on, both to gate what is being merged into them and the right point to merge them into a target release branch. If this is done, then these types of branches can be an important mechanism to ensure the overall

quality of the final product. If this is not done, then they can become just another bottleneck in the development and release cycle.

If the project is small enough in the scope of changes, the targeted release and production branch can be used as the integration branch as well—again with careful oversight. An even better approach is to utilize a tool such as Gerrit that essentially takes the place of an integration branch by providing an intermediate *holding area* for changes targeted for the remote repository. At the same time, it provides a convenient framework for doing best practices such as code reviews, verification builds, and so on, on the changes before they are merged into the remote repository.

In any of the branching workflows discussed in the following sections, if an integration branch isn't specifically mentioned, one can be added if needed—or, as a better solution, utilize Gerrit to provide additional benefits.

Release and Production Branches

As the name implies, release and production branches are branches with the final code to build the products that will go to customers. The main point about these branches is that they should be subject to the strictest oversight with respect to the gates for code being merged (or directly pushed) into them. Also, they should always go through the full checks for testing, quality metrics, and so on.

Using Tags in Branches

Another useful construct when working with branches is to identify key points in the lifecycle of a release by the use of tags. As I discussed in Chapter 7, a tag is simply an identifier used to point to the SHA1 value of a commit. And, unlike a branch identifier, the pointer stays with the tagged commit. The SHA1 value is already identifying the commit, but the tag is (or at least should be) much more user-friendly, easier to locate in the history, and easier to remember.

The set of key points to tag is generally up to the product teams. However, some suggestions would include integration points when features are merged into a branch—using a tag that incorporates an identifier for the feature—and points when particular groups believe their work is done, such as *development complete*. For added security and verification, a signed tag can be used, as discussed in the Advanced Topics section of Chapter 7.

The prime case for using tags in branches occurs when tagging release points where multiple releases are done from the same branch, one after the other. In that scenario, tags become especially important at identifying the branch point for creating separate branches if updates are later needed to a previous release. This is one of the workflows I'll discuss in the next section.

Branching Workflows

Separate from the use of branches to develop features or work on topics, there are multiple strategies for managing release branches. Each of these approaches has its advantages and disadvantages. I will cover these briefly as there are many more references available on the web that discuss these strategies—although different names may be used for the models. I encourage you to explore more sources and give careful consideration to which strategy may work best in a given situation before adopting one.

> **NOTE** *If you want to learn more about commonly used branching models with Git, you have only to search on the web. In particular, search for git flow.*

Master-as-Production

In the master-as-production model, releases are targeted to be from one designated branch, usually master. The idea is that, as development for one release is completed, a tag is created to mark the release point. Then, work continues in the same branch for the next release. If a need arises to update a previous release, a branch is created based on the tag that was created to mark that release. Then, the updates are completed and released out of that second branch (the one created from the tag).

This model can work well when development only needs to occur for one release at a time, such as in a true continuous delivery model. It has the advantage of limiting the number of release branches and only creating other release branches when an update or fix to a previous release is needed. This model is also more straightforward for users and processes, because they don't have to change environments or context from one release to the next. Another significant advantage is that when work is spread across multiple Git repositories, the repositories will all have a master branch by default, and so initial coordination to create a branch across all the repositories isn't required.

This model has the disadvantage of only facilitating working on one release at a time. As a result, it requires additional setup and context switching when you need to make an update to a previous release or work on multiple releases at once. Creating additional branches in Git is simple, but if handling for the additional branches needs to be set up through the build, test, and deployment pipeline, that can have a non-trivial cost. Some of the cost can be mitigated by configuring the pipeline processes to be able to handle multiple branches from the beginning, and using techniques such as whitelists or blacklists or patterns in branch names to automatically handle additional branches when they come along.

Another potential disadvantage can arise if there is remaining work to be committed after development is done, but before the release is ready. For example, translation or documentation work may need to be put into the branch after development is done but before they start on the next release. To handle this, the development team can work in a feature branch on the next release until the other groups' work for the current release is completed in master. At that point, the feature branch can be merged back into master. Figure 8-12 illustrates this model.

Master-to-Release

In the master-to-release model, development occurs in a single branch (like the master-as-production model) until late in the cycle. At the point where work on the current release is nearing completion, and the development team is ready to begin working on the next release, a new branch is created for finishing the current release. Work for the current release is finished in the new branch while

development begins on the next release in the original branch. Updates to existing releases can be made in the branch specific to that release, with tags used to separate the updates from the original release.

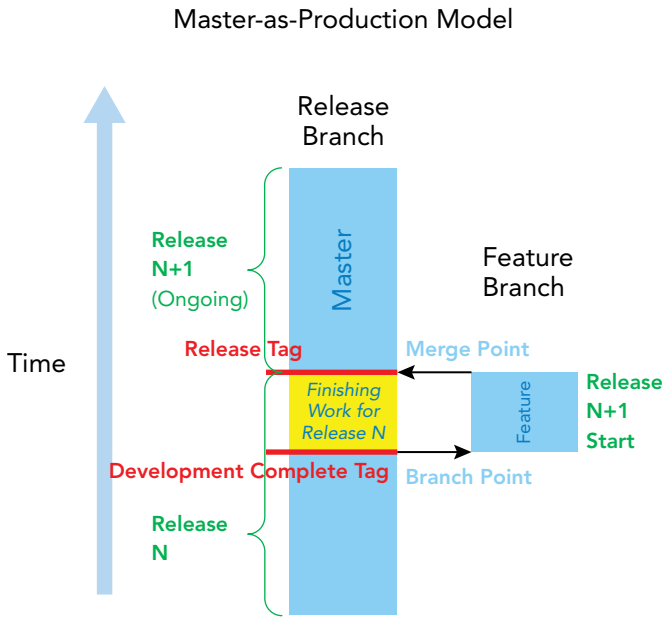


FIGURE 8-12: The master-as-production model

This model has the advantage of using a separate branch for each release, which allows for an expected place to handle finishing work and updates for each release without impacting work on the next release. It better supports concurrent development of multiple releases. Assuming the branch names are done in such a way as to map to releases, it can be easier to identify where code for a particular release exists, as opposed to tags in a single branch. This can be especially true if developers are using IDEs that understand how to work with branches implicitly but not with tags.

There are three disadvantages to this model. One is that, if a product is released frequently, the number of branches can quickly become unwieldy. While there is little overhead to Git, interfaces and processes that need to be aware of or work with each branch can get confusing. A second disadvantage is that there has to be a *context switch* at the late branching point for users and processes. This has to be coordinated to make sure that all groups and processes are aware and don't accidentally continue working in the original branch. The final disadvantage is that if work is happening across multiple Git repositories, they all need to have the new branch created in them to match and coordinate.

The last two disadvantages can be mitigated by appropriate planning and coordination in advance of the branching. Additionally, automating the process to create the branch based on a trigger event

and signal, and sending automatic notifications when that happens, can provide significant benefits. Figure 8-13 illustrates this model.

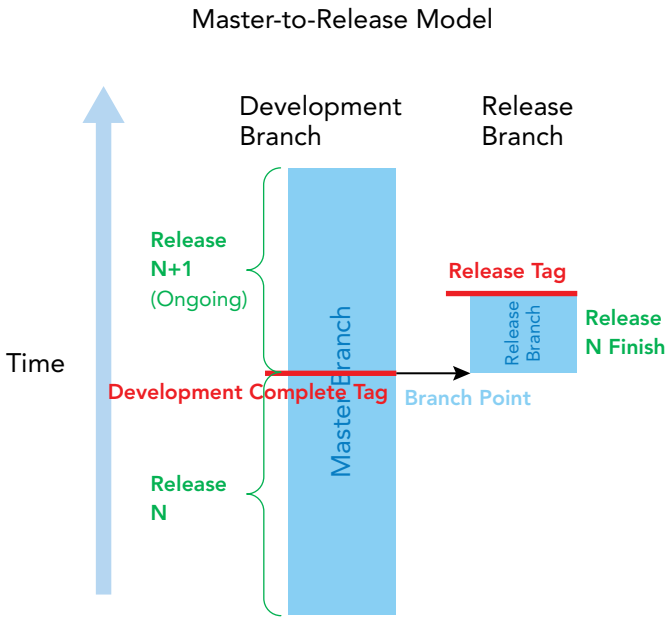


FIGURE 8-13: The master-to-release model

Master-as-Integration

A variation on the master-to-release model is to use the master branch as an integration branch—merging development from other branches into it, but not delivering from it. A separate release branch is used for releasing the code.

The advantage here is that there is a *buffer* between the feature branches and the release branch where integration work can be done. This avoids destabilizing the release branch if a lot of changes, or changes that are significant in scope, need to be regularly merged in. Also, if master is used by convention as your integration branch, then you already have the integration branch existing in all Git repositories.

The disadvantages of this model are that code may be *delayed* in the integration branch inordinately long if other work it uses or depends on is broken, or if there is a lot of manual review that has to happen.

The separately created release branch can be seen as either an advantage or a disadvantage for reasons I have already cited. Setup of the release branch can be handled in one of two ways:

- It can be created at (or near) the start of a release (early branching) and periodically have the integration branch merged into it throughout the development cycle.
- It can be created near the end of the release (late branching) from the integration branch, primarily for tasks related to finishing the release.

Another advantage is that there is a named branch associated with each release and the integration branch (master) doesn't have to be frozen; integration changes for the next release can continue to be merged into it while the secondary branch is finalized for production. At that point, the regular merges of the integration branch into the branch for the pending release are stopped. If a critical fix is needed in the release branch, it can be done directly in that branch.

For teams considering using an integration branch, Gerrit should also be explored as an alternative. Figure 8-14 illustrates this model.

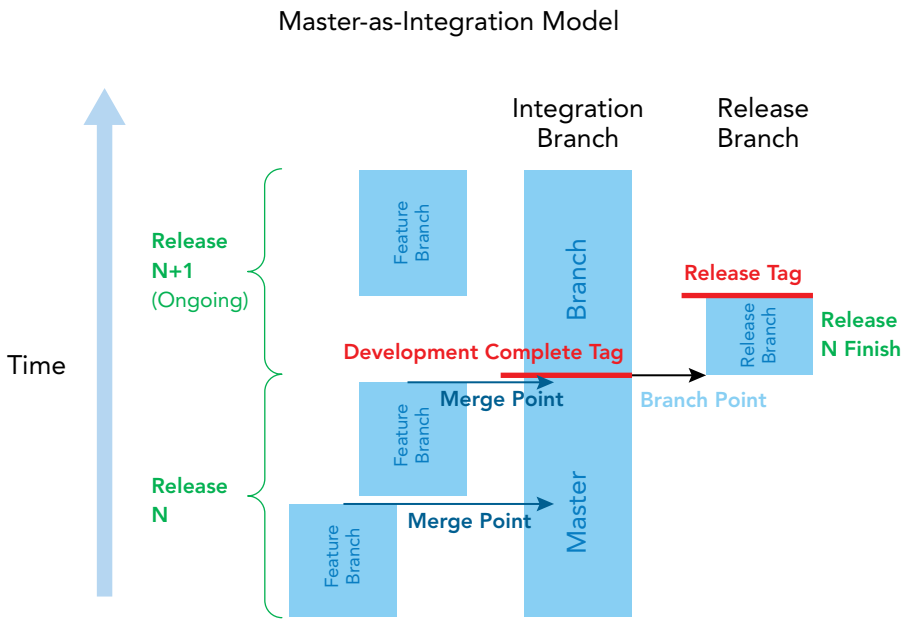


FIGURE 8-14: The master-as-integration model

It's worth noting that in the models where you use master as an example, that could be any designated branch, as long as the use is consistent across the multiple Git repositories if a product consists of multiple repositories. Because master is always present by default and is the default of some commands, its use is typically more convenient than other branches in these types of models.

Parallel

In the parallel model, a separate branch is created for each release when work on that release is begun. Work for that release is done in that specific branch throughout the lifecycle of the release.

This model has the advantage of providing a designated and separate environment from the start for each release. Thus, work on multiple releases can proceed in parallel as early as needed. No context switching is required once environments are initially set up.

A disadvantage of this model is arguably the number of branches created over the course of a product's releases. As well, setup costs (beyond Git branching) multiplied by the number of releases may become prohibitive. Figure 8-15 illustrates this model.

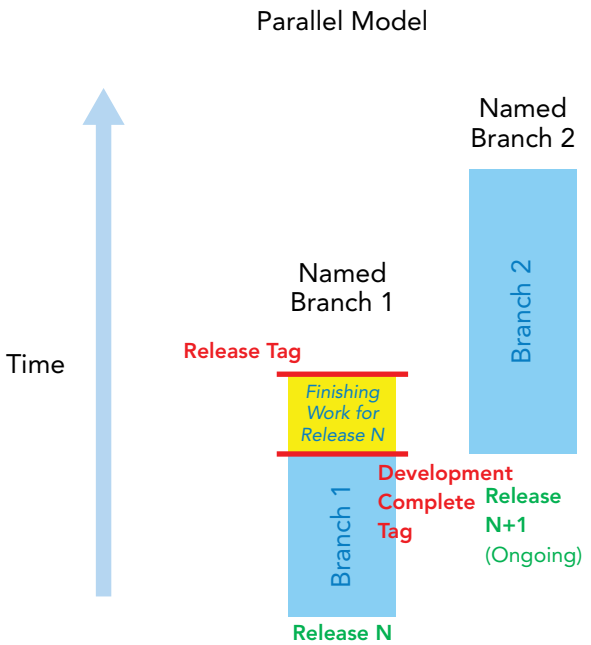


FIGURE 8-15: The parallel model

Conclusion

From reading this chapter, you should gather that there are many factors to consider before implementing a branching model. In particular, depending on the frequency of releases, using multiple branches may or may not be an issue. As noted, normally the Git overhead is not a significant factor (tags and branches are both pointers). It is usually the supporting pieces and interfaces, as well as user preferences, that dictate which model will work best.

In all cases, topic and feature branches are useful when work needs to be done outside of the production and release branch and then merged in if appropriate.

A final key point here is that Git makes changing models fairly easy, given how simple it is to create branches from existing ones. Another key factor is the ease of merging content from one branch to another. I'll discuss this in Chapter 9.

ADVANCED TOPICS

In this section, I'll discuss using checkout and branch with SHA1s that aren't already associated with a branch. Related to that, I'll look at some particularly unnerving terminology that Git will sometimes throw at you.

Also, I'll discuss what it means to depart from the usual model of working with an entire commit and to check out individual files, and what the implications are of that approach.

Checking Out Non-branch Commits

As I've mentioned previously, branches and tags in Git are really just pointers or references to particular SHA1 values of particular commits. Internally, in its .git structure, Git is storing the SHA1 values under refs/heads/<name> for branches and under refs/tags/<name> for tags.

Because branch and tag references are translated to the SHA1 values, this supports a useful feature for commands that accept branch and tag references as arguments. These commands can accept any SHA1 value of a commit stored in Git as an argument, whether or not that SHA1 value is mapped to a branch or tag.

You've already seen this in Chapter 7, when I mentioned being able to pass an *absolute* SHA1 value to the *reset* command. Refer back to Figure 7-4 for the illustration of this.

In the reset case, I used this feature as a kind of *rollback*, moving the HEAD pointer back to a previous commit and optionally updating the staging area and working directory. At that point, I hadn't covered branches, but adding branches into the equation, using reset to move the HEAD implies that the pointer for the current branch is also moved. Running a log on master before and after the command shows the results of the move.

Before the reset:

```
$ git log --oneline master
d21be2c third change
43bd3ef second change
87ba8bc first change

$ git reset --hard 87ba8bc
```

After the reset:

```
$ git log --oneline master
87ba8bc first change
```

> **NOTE** *If you want further verification that the branch has been reset when you reset HEAD, you can look in the filesystem. For example, to see what master points to, you can use the command:* `cat .git/refs/heads/master`.
>
> *This returns the SHA1 value starting with the result of the reset:*
>
> ```
> 87ba8bcff6be9cfea0e3dd0feb2407eebc2dc072
> ```
>
> *You can also see that HEAD points to master (so they* move *together while master is the current branch) by using the command* `cat .git/HEAD`. *In this case this will return* `ref: refs/heads/master`.

Extending this to checkout, you can also check out any SHA1 value associated with a commit in the repository.

Why would you want to do this? Consider a case where you need to go back and get a previous version of code to validate functionality of something that is currently broken. Or, as referenced earlier

in this chapter, you might need to go back and start a new branch off of a known point in the past, before other changes were introduced.

Let's look at an example of the mechanics of this process. Suppose you have a repository with a branch named *feature1* with four commits.

```
ca27770 fourth change
31de2b4 third change
fc28c0d second change
25c56c4 first change
```

You could do a checkout on the second commit using the command: `git checkout fc28c0d`. When you run a command like this one, Git performs the command, but returns a rather ominous-looking message.

```
Note: checking out 'fc28c0d'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b <new-branch-name>

HEAD is now at fc28c0d... second change
```

I'll discuss the implications of this message in the next section.

Dealing with a Detached HEAD

Although frightening at first, this message is actually helpful if you read through it. Essentially, Git is pointing out that you have checked out a committed revision that is not the most current one on the branch. HEAD now points to that commit instead of the branch. And the message is explaining your options.

Prior to your checkout command, the repository looked like Figure 8-16. (Note: For simplicity, I am not showing the master branch here.)

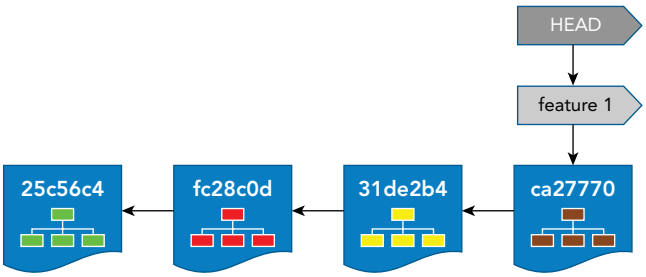


FIGURE 8-16: Repository before checkout of fc28c0d

After the checkout of the second commit, the repository looks like Figure 8-17. (Again, master is omitted.)

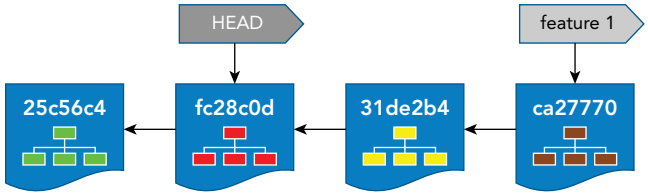


FIGURE 8-17: Repository after checkout of fc28c0d

If you now run a *git log* command, it defaults to HEAD and you see output like this:

```
machine name> ((fc28c0d...))
$ git log --oneline
fc28c0d second change
25c56c4 first change
```

Notice that first line before the log command—your system name and prompt. This assumes you have your prompt configured (as discussed earlier) to show the current branch—or HEAD in this case because you're in the *detached HEAD* state. And if you do a log of (or from) feature1, you see the full chain of commits as expected.

```
machine name> ((fc28c0d...))
$ git log --oneline feature1
ca27770 fourth change
31de2b4 third change
fc28c0d second change
25c56c4 first change
```

As the message indicated when you checked out the specific commit, you can do essentially anything in this detached state, based on this commit, that you could normally do on a branch. Let's assume you make a change from this point and stage it. If you were to check the status here, Git would helpfully remind you again that you're in the *detached* state.

```
$ git status
HEAD detached at fc28c0d
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   <filename>
```

Now, you can go ahead and commit the change. Note the output message.

```
$ git commit -m "update based on second change"
[detached HEAD 2a93f89] update based on second change
 1 file changed, 1 insertion(+)
```

Your repository now looks like Figure 8-18.

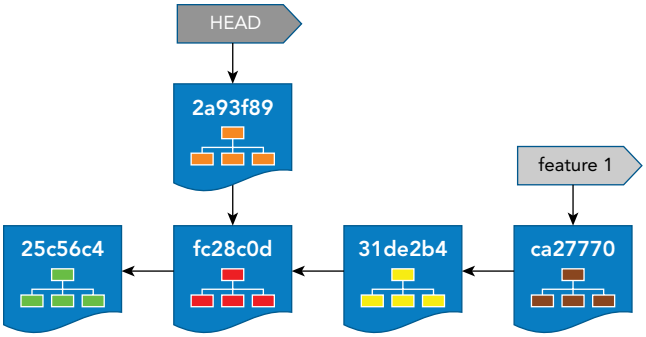


FIGURE 8-18: Repository state after the new commit

Now, running a log command that defaults to HEAD shows the path from HEAD to the first commit.

```
2a93f89 update based on second change
fc28c0d second change
25c56c4 first change
```

And the log for feature1 still shows the original chain that it points to.

```
ca27770 fourth change
31de2b4 third change
fc28c0d second change
25c56c4 first change
```

After you're done working off of this commit, you can switch back to your original branch using the checkout command. Notice the message that Git returns when you do this.

```
$ git checkout feature1
Warning: you are leaving 1 commit behind, not connected to
any of your branches:

  2a93f89 update based on second change

If you want to keep it by creating a new branch, this may be a good time
to do so with:

 git branch <new-branch-name> 2a93f89

Switched to branch 'feature1'
```

This message is pretty self-explanatory. Your repository currently looks like Figure 8-19.

Git is telling you that you don't have any references that refer to the new commit you made while in detached HEAD mode. You haven't lost the commit—it is still there. However, in order to access it, you have to know the SHA1 value associated with it, rather than a more user-friendly reference such as a branch.

In case you want to continue this line of development later, let's go ahead and create a branch off of that commit. This form of the branch command demonstrates how to create a branch off of a commit SHA1. However, you could also use any other branch name as the last argument to create a new branch off of a branch that is not the current one. The command is `git branch experimental 2a93f89`. After this command, your repository looks like Figure 8-20.

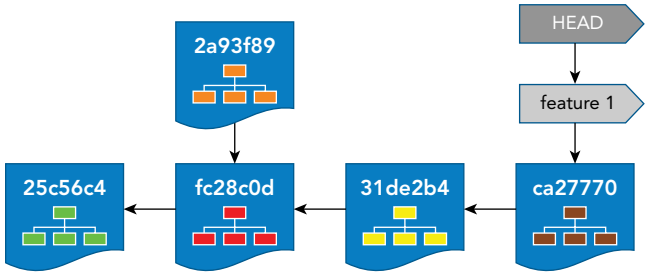


FIGURE 8-19: Repository after you switch back to feature1

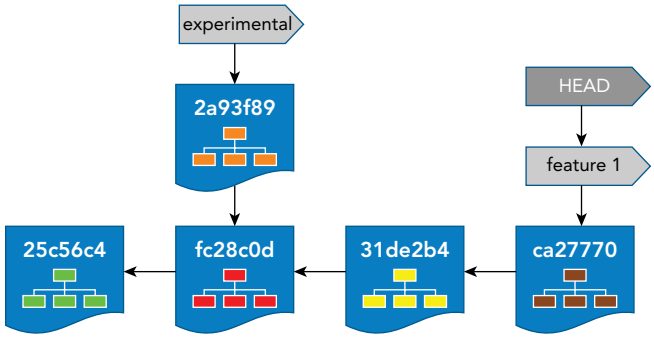


FIGURE 8-20: After creating a new branch off of your commit

You can now run any of your usual commands on the new *experimental* branch, such as log and checkout.

```
$ git log --oneline experimental
2a93f89 update based on second change
fc28c0d second change
25c56c4 first change

$ git checkout experimental
Switched to branch 'experimental'
```

After a checkout to make *experimental* the current branch, the repository looks like Figure 8-21. (And, of course, your working directory is populated with the contents of *experimental*.)

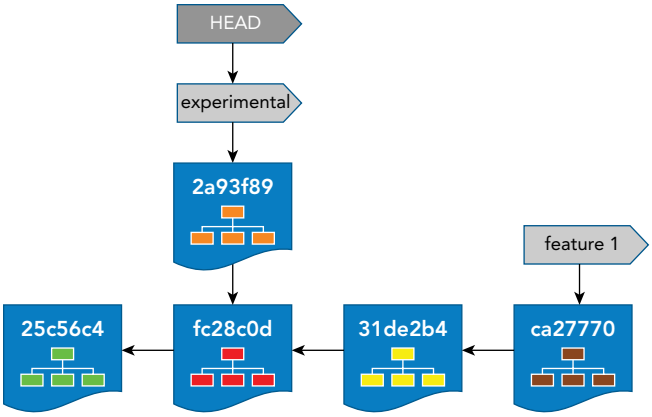


FIGURE 8-21: After a checkout of experimental

Figure 8-22 shows the two paths you now have through your repository for the two branches. These are the series of commits you see in a log command output for the respective branches. (Master is also there, but is not shown.)

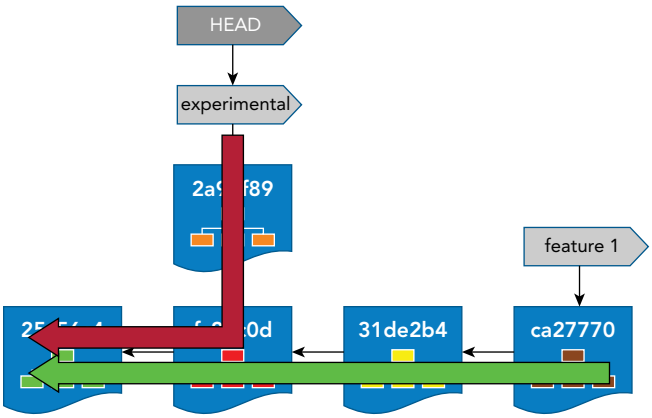


FIGURE 8-22: The two paths of your two branches

Checking Out Individual Files

Normally with Git, you are working at the scope of a commit—a snapshot of an entire tree. But, as I note in Chapter 5, it is possible to use many operations with individual files. Checkout is no exception.

There are two basic forms of the checkout command that you will probably want to use when working with individual files: checking out the latest version and checking out a version from a specific commit.

Both of these forms are straightforward. Checking out the latest version is just a matter of running this command: `git checkout <filename>`.

Checking out a version from a specific commit just requires adding the reference (branch, tag, and so on) or the explicit SHA1 value of the commit to get the file from. The command would be of this form: `git checkout <reference | SHA1> <filename>`.

Note that you can also pass multiple filenames on the command line or use names with wildcards to check out multiple files at once.

> **NOTE** *Occasionally you might try to check out a file and get an error message like this:*
>
> ```
> error: pathspec <filename> did not match any file(s) known to git.
> ```
>
> *This usually occurs for one of two reasons:*
>
> 1. The filename is spelled incorrectly (typo).
> 2. The filename is valid but does not exist on whatever revision you're trying to check it out from. An example would be that a file was added on the third commit to a branch, but you are trying to check it out from the second commit. Or you are trying to check out the file from a branch where it does not exist in the branch.
>
> *Validating these two situations will usually identify the cause of the error.*

Unlike checking out an entire commit, checking out individual files does not affect the current branch. You just end up with a different version of the particular files in your working directory. Be aware, though, that this operation overwrites the files in your working directory, so be sure to commit or save a copy if you haven't already done that and want to preserve that version.

SUMMARY

In this chapter, I introduced the concepts and mechanics of working with branches in Git. Branching in Git is very quick and easy and is one of the key features that attracts users to Git. Because of the way Git stores changes internally, a branch is just a named pointer to a particular SHA1 value and commit. This pointer is moved as content is added, merged, or removed, unlike the pointer for a tag, which always stays with the commit that was tagged.

Checking out a branch switches to that branch and also updates content in the working area to be consistent with the branch. This allows for another powerful feature of Git: being able to work in one working area per repository, regardless of how many branches are used. Switching branches does assume that there are no uncommitted changes in the working directory.

There are some common workflows for using branches with Git. Many of these workflows revolve around the idea of only using one branch (usually master) and doing multiple releases out of that branch, tagging release points in between.

In the next chapter, I'll talk more about the various ways that Git allows you to merge branches together.

About Connected Lab 5: Working with Branches

In this lab, you'll get an opportunity to set up a branch and explore how the checkout command allows you to switch between branches and content in the working directory.

You'll also get a chance to play with the graphical interface gitk that comes with most versions of Git.

Connected Lab 5

Working with Branches

In this lab, you'll start working with branches by creating a new branch and making changes on it.

PREREQUISITES

This lab assumes that you have done Connected Lab 4: Using Git History, Aliases, and Tags. You should start out in the same directory as that lab.

STEPS

1. Starting in the same directory that you used for Connected Lab 4, use the git branch command to look at what branches you currently have.

   ```
   $ git branch
   ```

2. You see a line that says "* master". This indicates that there is currently only one branch in your repository: master. The asterisk (*) next to it indicates that it is the current branch (the one you've switched to and are currently working in). If your terminal prompt is configured to show the current branch, it also says "master".

3. Before you work with a new branch, you need to update the files in the master branch to indicate that these are the versions on master so it will be easier to see which version you have later. To do this, you can use a similar version of the same way you have been creating and updating other files. Run the following command for each file.

   ```
   $ echo "master version" >> filename
   ```

4. Stage and commit the updated files. Because these are files that Git already knows about, you can use the following shortcut command:

   ```
   $ git commit -am "master version"
   ```

5. When you work with branches, it can be helpful to see a visual representation of what's in the repository. To do this, you can use the gitk tool that comes with Git. Start up gitk in this directory and have it run in the background.

```
$ gitk &
```

6. In gitk, create a new view, as follows:

   a. In the menu, select View, and then select New View. A dialog box will open. In the dialog box, find the field named "View Name". Type in a new name for this view.

   b. Check the "Remember this view" check box.

   c. Check the four check boxes under the "Branches & tags:" field. Click OK.

   d. If needed, switch to the new view under the View menu.

7. You have a new feature to work on, so you now create a feature branch with the name *feature*. Switch back to your terminal, and in the directory, run the following command:

```
$ git branch feature
```

8. Notice that this command creates the branch, but does not switch to it. You can now check what branches you have and which is your current branch.

```
$ git branch
```

9. You can now see your new branch listed. Change into the feature branch to do some work:

```
$ git checkout feature
```

10. To verify that you're on the feature branch, run the following command, and observe that the asterisk (*) is next to that branch:

```
$ git branch
```

11. Switch back to gitk, and refresh the screen, or reload, to see what the window showing the branches, tags, and changes looks like now. You should see your new branch showing up there now.

12. Back in the terminal session, create a new file and then update the files in the feature branch to indicate that they are the *feature branch version*.

```
$ echo "new file" > file4.c
$ echo "feature version" >> filename (for each file)
```

13. When you're done, stage and commit your changes.

```
$ git add .
$ git commit -m "feature version"
```

(Note: If you just use `git commit -am`, it doesn't pick up your new file.)

14. Refresh/reload your view in gitk and take one more look at your feature branch.

15. In your terminal session, switch back to the master branch.

```
$ git checkout master
```

16. Verify that you're on the correct branch.

    ```
    $ git branch
    ```

    (Note: The branch listing should have an asterisk [*] by master.)

17. Look at the contents of the files and verify that they're the original ones from master.

    ```
    $ cat filenames
    ```

18. Look for "master version" in the text.

19. Refresh/reload gitk and note any changes in the upper left window.

9

Merging Content

WHAT'S IN THIS CHAPTER?

- The Git merge operations
- Types of merges that Git supports
- Merge strategies
- Rebasing
- Cherry-picking
- Visual merge tools
- Different ways to show conflicts
- Advanced rebasing
- Interactive rebasing

This chapter introduces the Git model of merging branches as well as the merge types that are supported in Git. I also discuss the Git rebase functionality—a sort of merging with history—and the more selective operations that allow you to merge in an individual commit. Finishing out the main part of the chapter, I discuss how Git deals with conflicts and how you can resolve them.

In the Advanced Topics section, I discuss some advanced rebase functionalities, including *interactive rebasing* that allows for updating a set of commits already in the repository.

THE BASICS OF MERGING

As you've previously seen, Git stores changes at the granularity of a commit—a snapshot of your entire working directory tree at a point in time. You've also learned that each commit has the potential to become a branch and that a branch in Git is really just a named, moving pointer that points to a commit, which, in turn, points to other commits in the chain.

Given that a branch points to a commit and that a commit is the granularity that you associate changes with, it shouldn't come as a surprise that when I talk about merging in Git, I'm talking about merging at the branch or commit level. Thus the command for merging assumes you are dealing with a SHA1 value or a name that references a SHA1 value (branch or commit) as the things to merge.

The Merge Command

The syntax for doing a merge is as follows:

```
git merge [-n] [--stat] [--no-commit] [--squash] [--[no-]edit]
          [-s <strategy>] [-X <strategy-option>] [-S[<keyid>]]
          [--[no-]rerere-autoupdate] [-m <msg>] [<commit>...]
git merge <msg> HEAD <commit>...
git merge –abort
```

The simplest form to work with is `git merge <branch name>`.

In this form, the command is attempting to merge `<branch name>` into your current branch. Other forms provide more flexibility, including merging more than two things. However, for your purposes, this simple form will suffice in most cases.

Merge situations typically result from two or more users, or lines of development, trying to update common code. Most commonly, in a local environment, a user is trying to merge a branch into another branch in their local repository.

Recall that I said that the workflow in Git is typically branch-based, with Git empowering users to create branches for practically any purpose, from creating bugs or fixes to experimental code. As I discussed in Chapter 8, at some point, these targeted, individual branches may be deemed good enough to be merged into an integration branch, if one is used. Then at some point, after further testing and updates, the integration branch may be merged into a production branch. And so the cycle continues.

Preparing for a Merge

Git is not able to deal with any uncommitted changes when starting a merge. If you have changes in the working directory or in the staging area that have not been committed, and you try to do a merge, Git responds with a message like this:

```
Updating <commits>
error: Your local changes to the following files would be overwritten by merge:
       <file names with uncommitted changes>
Please, commit your changes or stash them before you can merge.
Aborting
```

The *stash* command is a way to save the state of any uncommitted changes from your working directory and staging area in a *stash* off to the side. The stash command is discussed in detail in Chapter 10.

The idea is that using one of these methods, either committing the changes or saving them off in a stash, you end up with a clean working directory and staging area—no uncommitted changes. That way, if the merge is successful, Git can update the local repository with no *dirty* (modified but not committed) files left behind.

Types of Merges

Depending on the options specified, Git can attempt to do a merge in one of several ways. I'll discuss some details about these different types of merges in the following section. I'll also use some illustrations to help you understand what happens in each type.

Fast-Forward—an Optimization

Fast-forward is the default merge behavior in Git. A fast-forward merge can happen if what you're trying to merge in already contains all of the content of the destination, and your changes are just additions beyond that.

Put another way, there have not been any other changes made to the *destination* branch that you don't have in the *source* branch; however, the source branch has newer changes to be *appended*. This implies that the branch you're trying to merge into needs to be a direct ancestor of the branch you're merging from.

In this case, Git can do an optimization. The optimization here is that it can just move the branch pointer of the destination branch to the latest commit you're merging in. It can, in effect, *fast-forward* the pointer instead of having to do an actual merge.

Think of this like watching a recorded show on your DVR or listening to a piece of music on a player. There is a straight stream of content, and if you fast-forward, you're just advancing the pointer to a new part of the same stream that already has the parts you've seen or listened to.

Figures 9-1 and 9-2 illustrate how a fast-forward merge works. In Figure 9-1, I start with a sequence of commits with two branch points. The original set of changes that includes C1, C2, and C3 was committed on branch *master*. Then branch *fix* was created and C4 and C5 were committed on it.

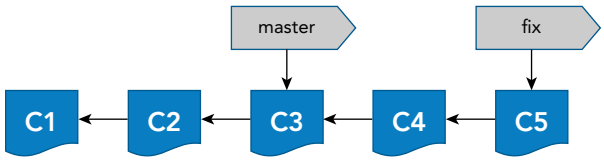


FIGURE 9-1: Setup for the fast-forward example

Taking a look at the figure, it's easy to see the relationship between the two branches. *Fix* contains all of the commits of *master* plus the two additional ones.

If you create logs of the contents of the branches, you can see how the commits map to them. (For clarity, I've used the names in the figure as the comments for the commits.)

```
$ git log --oneline fix     $ git log --oneline master
cb323c7 C5                  31adc9b C3
eb24943 C4                  336715c C2
31adc9b C3                  7058e67 C1
336715c C2
7058e67 C1
```

Suppose you now want to merge *fix* into *master*. In the simplest form, you would first check out the destination branch, *master*, and then merge in *fix*. The commands are straightforward: *git checkout master* followed by *git merge fix*.

When you issue these two commands, Git notices that branch *fix* already has all the commits of branch *master* in its chain. This means that no actual merging is required. If you actually performed merge actions, you would still end up with the contents of C5 after the merge. So, Git recognizes that if it just moves the pointer for *master* to the same commit as *fix*, it makes *master* include C5 (and C4). As a result, it does the optimization and just moves (*fast-forwards*) the branch pointer. The output of executing these commands looks like this:

```
$ git checkout master
Switched to branch 'master'

$ git merge fix
Updating 31adc9b..cb323c7
Fast-forward
 file1.txt | 2 +-
 1  file changed, 1 insertion(+), 1 deletion(-)
```

Notice the reference to *fast-forward* in the text. Figure 9-2 shows the results.

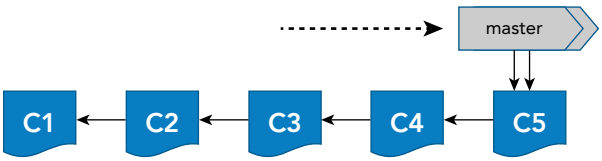


FIGURE 9-2: The fast-forward merge

After the merge is completed, the logs for both branches look like this:

```
cb323c7 C5
eb24943 C4
31adc9b C3
336715c C2
7058e67 C1
```

Three-Way Merge

The second type of common merge that users encounter in Git can be referred to as a *three-way merge*. In this scenario, a fast-forward is not possible because both branches to be merged have updates since they diverged from a common point. Those divergences have to be resolved before the merge can be completed.

Here, Git uses more of a traditional merge strategy. Git looks at three pieces of information: the tip (most recent commit) of the source branch, the tip (most recent commit) of the destination branch, and the common ancestor (last commit that they had in common). From these three commits, Git interpolates how to do a merge and attempts it. If successful, Git creates a new commit—the results of the merge, also called a *merge commit*. Figures 9-3 to 9-6 illustrate this process.

In Figure 9-3, you start with a sequence of commits with two branch points. Commits C1 and C2 were made to branch *master*. Then branch *feature* was created and C3 was committed on it. Commit C4 was made on branch *master* and then C5 was made back on branch *feature*.

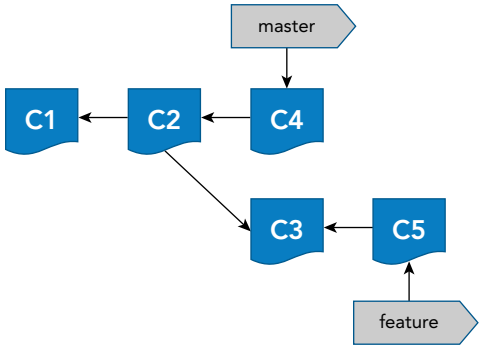


FIGURE 9-3: Setup for the three-way merge example—not eligible for fast-forward

Git evaluates the situation and determines that a fast-forward merge is not possible, because both branches have changes since the common ancestor. Git first identifies the three commits that it will use in the merge: the ends of the branches and the common ancestor. These three points are highlighted in Figure 9-4.

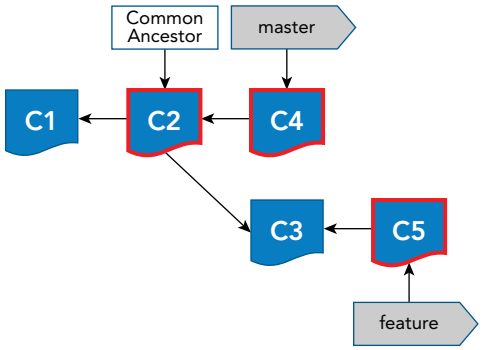


FIGURE 9-4: The three points considered for the three-way merge

Then Git tries to do the merge, using the contents of these three commits to decide how to process differences and which changes should or should not be carried forward. Figure 9-5 illustrates this process.

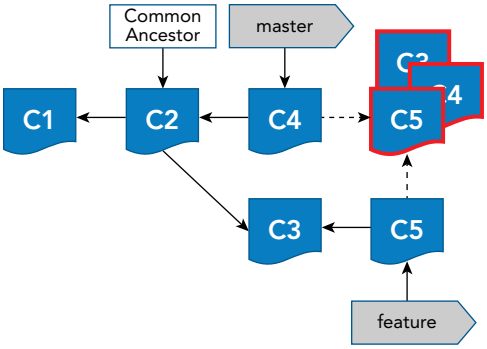


FIGURE 9-5: The three-way merge process

If the merge is successful, you end up with a new *merge commit* (the result of the merge), C6, as shown in Figure 9-6. And, because you were merging into branch *master*, the master pointer has been advanced to the new commit—effectively updating the contents of branch *master* with the results of the merge.

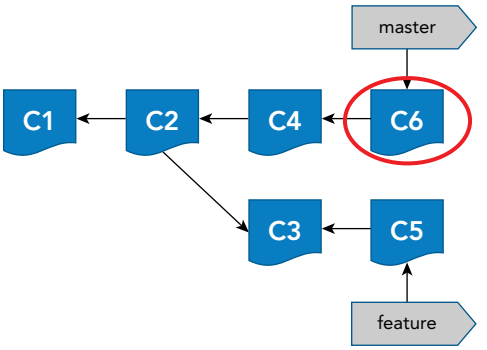


FIGURE 9-6: The new merge commit after the three-way merge

> **NOTE** *Git merge defaults to trying to do a fast-forward merge first. If it can't do that, it does a three-way merge and creates a merge commit. However, there are options to the merge command that allow the user to specify a particular merge behavior with respect to fast-forward merges.*
>
> *The* --ff *option tells Git to do a fast-forward merge if possible. This is the default.*
>
> *The* --no-ff *option tells Git not to do a fast-forward merge. In effect, this tells Git to create a merge commit even if it could complete the operation via a fast-forward merge. One reason to use this option would be to preserve a clearer history of when a branch was merged in (as evidenced by the creation of the merge commit), and a clearer trail of the history of the branch.*
>
> *The* --ff-only *option tells Git to only do the merge if a fast-forward can be used.*

Note that the examples I have presented here show what happens in the local repository. The merging happens in the working directory, and, if the results are successful, then the merge will be committed automatically. If you prefer instead to review the results in the working directory and commit the merged content yourself, you can add the *--no-commit* option to the command.

As another refinement, you can also add the *--edit* option to have Git stop before doing an automatic commit and allow you to add more details to the auto-generated merge message. This fits in with the philosophy I discussed in the earlier chapters of having meaningful commit messages that others can understand easily.

Rebasing—Merging with History

Within Git, there are two main ways to incorporate a set of changes between branches: merging and rebasing. Although the endpoints after a merge or rebase should be effectively the same, the process and history associated with the two operations are very different.

As I previously discussed, in a three-way merge scenario, Git looks at the current endpoints of the two branches and their last common ancestor. From those three commits, it attempts to construct a new commit—a *merge commit*.

In a rebase scenario, Git identifies the last common ancestor, and, for each change beyond that ancestor in the source branch, it proceeds as follows:

- It computes the *delta* of that change—what changed with that commit.
- For each change, it attempts to *replay* (apply) that delta onto the branch you're merging into. Another way of thinking about this is that it attempts to make the same set of changes on top of the latest commit in the destination branch.

If all of these deltas apply cleanly (without conflicts), then the end result is that the unique changes from the one branch become a part of the linear history of (are appended to) the other branch.

So why do a rebase? The endpoint (last merged commit) for a rebase and a merge should be effectively the same. A merge process is happening in both cases, leading to an endpoint commit. However, in a rebase, you build up a history. Individual merges (applying each delta in turn) are done along the way to create a linear history of changes from one branch in another branch. This, in turn, makes for a cleaner and more detailed merge sequence. It includes the progression of changes from the other branch that led to the new endpoint. This is as opposed to just the modified endpoint itself, which is what you get with a three-way merge.

WARNING *You should not rebase content that has already been pushed to a remote repository! The reason for this is the potential impact it could have on others trying to merge in their changes on the remote side afterward. Consider the following scenario:*

- *You make a set of changes locally in a branch, stage, commit, and then push those changes out to the remote (server) side (remote repository).*
- *Another team member pulls down these changes and starts making other changes on top of them.*

continues

continued

- *You then use the rebase command and rebase this branch on another one—significantly adding to the history of the branch.*
- *You commit and push your branch with the rebase changes up to the remote.*
- *The other team member finishes their changes, commits them, and tries to push them. They then encounter significant merge conflicts because the history has been changed.*
- *They then try to update their latest content to incorporate the rebased versions and the changes they want to make. However, depending on how much content was rebased, this may be a significant change and disruption for them, requiring a lot of time to resolve.*

I will talk more about this kind of scenario in Chapter 13 when I discuss remotes and remote branches.

If you absolutely must do a rebase (or other history-changing) operation on something that has already been pushed to the remote repository and used by others, a better approach is as follows:

- *Choose a future point in time to make the change.*
- *Communicate to anyone else using that repository that you will be making a change that alters history.*
- *Communicate to them that, prior to that point in time, they should have all changes committed* and pushed *from their side.*
- *Then make your changes at the appointed time, pushing them to the remote repository (as I will discuss in Chapter 12).*
- *Everyone else should then wipe out their cloned remote and clone a new copy with the updates.*

The syntax for a rebase is as follows:

```
git rebase [-i | --interactive] [options] [--exec <cmd>] [--onto <newbase>]
        [<upstream> [<branch>]]
git rebase [-i | --interactive] [options] [--exec <cmd>] [--onto <newbase>]
        --root [<branch>]
git rebase --continue | --skip | --abort | --edit-todo
```

To keep it simple, you can think of it in this default form: *git rebase branch2 [branch1]*.

Here, *branch2* is the destination branch, and *branch1* (or the current branch if *branch1* is not supplied) is the source branch for the updates that you will attempt to rebase to *branch2*.

You may find it useful to think of this command as follows: "Make Git append to the end of *branch2* the series of deltas from *branch1* since it diverged."

Let's look at an example. In Figure 9-7, you have two branches that have both diverged since the common ancestor: *feature* and *master*.

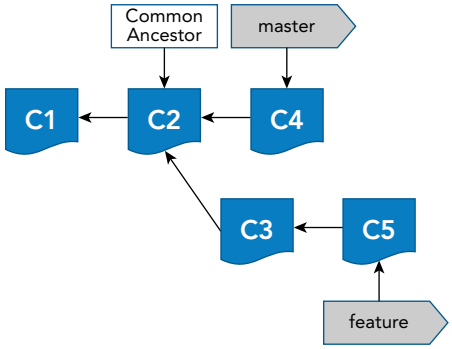


FIGURE 9-7: Setup for the rebase example

You've seen how to merge the tips of two branches like this earlier in this chapter. What you want to do in this case is to rebase feature onto master so that your feature branch contains the newest updates from master. You can use these commands: *git checkout feature* followed by *git rebase master*.

Git does the following:

- Goes to the common ancestor of the two branches (Figure 9-8)

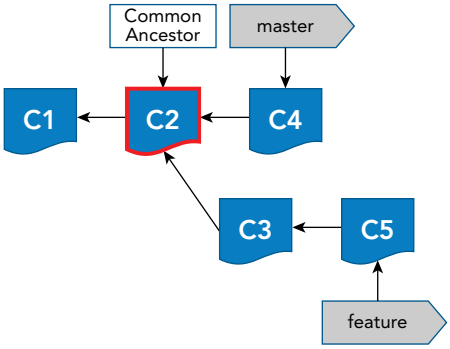


FIGURE 9-8: Identifying a common ancestor

- Computes the diff introduced by the commit of each change past the common ancestor on the current branch (or *branch1* if specified), and saves the diffs to temporary files (Figure 9-9)

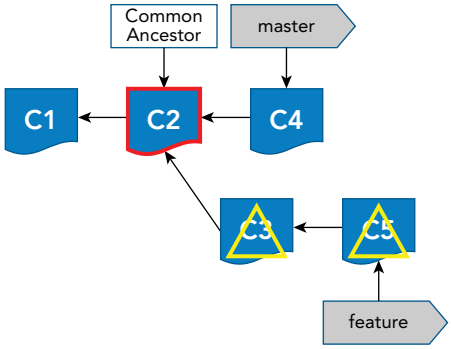


FIGURE 9-9: Computing deltas from the source branch

- Attempts to apply each change at the end of the destination branch in turn (Figure 9-10)

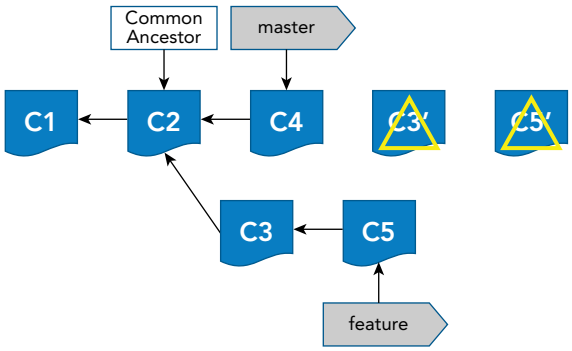


FIGURE 9-10: Applying deltas on the destination tip

If all of the rebasing is successful, then that feature will now have the latest changes from master incorporated into its history (Figure 9-11).

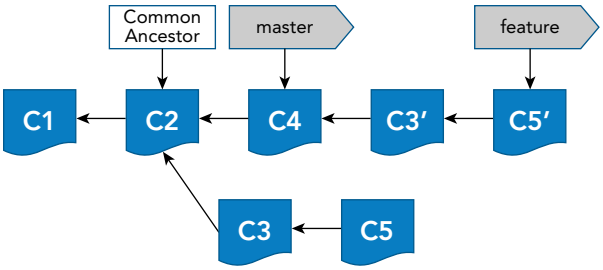


FIGURE 9-11: Completed rebase of a feature on master

This will also be visible via *git log* on that branch.

```
$ git log --oneline feature
71be157 C5
```

```
63d5b68 C3
1c72fc5 C4
18be5e0 C2
d7e08ad C1
```

The C3 and C5 in the log correspond to the C3′ and C5′ in Figure 9-11. Notice that the original C3 and C5 are still there, although no branch points to them any longer. Over time, those "orphaned" commits may be *cleaned up* by a garbage collection operation.

Cherry-Picking

Git includes one other type of primary merging operation: cherry-pick. The idea with this operation is to be able to selectively choose an individual or group of commits from within one branch and apply them to another branch. This command offers more fine-tuned selection than a general merge or rebase. Because the cherry-pick operation can be taken from any commit, it also requires more forethought and potentially more care when merging.

The syntax for cherry-pick is as follows:

```
        git cherry-pick [--edit] [-n] [-m parent-number] [-s] [-x] [--ff][-S[<keyid>]]
<commit>...
        git cherry-pick --continue
        git cherry-pick --quit
        git cherry-pick --abort
```

Because I have already covered merging and rebasing, the syntax and examples should look familiar at this point. Figure 9-12 shows a setup where you want to cherry-pick C5 from the feature branch into the master branch.

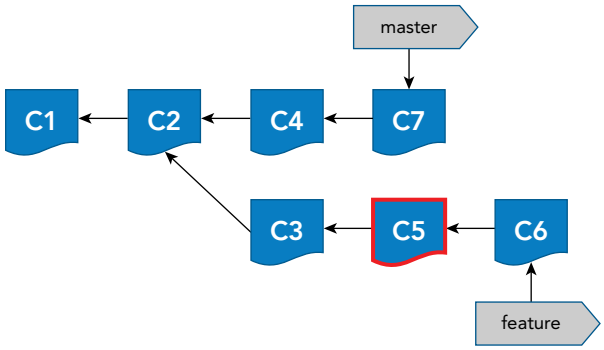


FIGURE 9-12: Setup for the cherry-pick example

Here are the logs for *master* and *feature* prior to the cherry-pick:

```
$ git log --oneline master      $ git log --oneline feature
34142bf C7                      818d819 C6
de6b1b2 C4                      d9e8b2c C5
d3b906a C2                      b942f21 C3
b4741c6 C1                      d3b906a C2
                                b4741c6 C1
```

Note that I have focused on C5 because that's the commit you want to cherry-pick. Because you need to reference that specific commit, and there is no branch or tag currently pointing to it, you would need to either reference it relative to an existing branch or just use its SHA1 value. I'll use the SHA1 value.

To actually do the cherry-pick for the intended revision, you enter the command, *git cherry-pick d9e8b2c*. The output shows that it worked.

```
[master 9f308b6] C5
 Date: Tue Jun 7 11:00:26 2016 -0400
 1 file changed, 1 insertion(+), 1 deletion(-)
```

The process that happens is similar to the rebase process for the one commit. Figure 9-13 shows the end result. (This assumes no merge conflicts.)

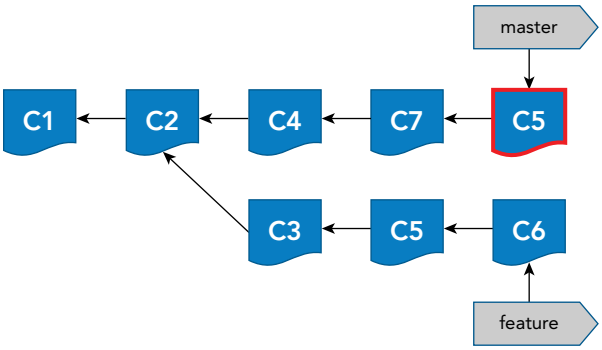


FIGURE 9-13: End result of the cherry-pick

You can now verify that C5 has been applied to the master branch via the log listings. Note that C5 in master and C5 in feature are different SHA1 values.

```
$ git log --oneline master      $ git log  --oneline feature
9f308b6 C5                      818d819 C6
34142bf C7                      d9e8b2c C5
de6b1b2 C4                      b942f21 C3
d3b906a C2                      d3b906a C2
b4741c6 C1                      b4741c6 C1
```

As this example illustrates, while I have been using branch names in the merge and rebase commands, it is also valid to use SHA1 values from other commits. The workflow and internal processing are the same—you are just pointing these operations to commits that aren't pointed to by a branch.

Specifying a Range of Commits

Note that it is also possible to specify a range of commits for a cherry-pick in the form of *<starting commit>..<ending commit>*. So, if you are starting before you cherry-pick C5, and instead want to cherry-pick the range of C3..C5 from feature (two commits, C3 and C5), it seems reasonable to specify the two SHA1 values directly as in this command: *git cherry-pick -xtheirs b942f21..d9e8b2c*. The "-xtheirs" option here specifies a resolution strategy to solve merge issues. It is discussed in more

detail in the section on "Resolution Options and Strategies" later in this chapter. It is not required to issue the cherry-pick command.

Here's the output you get:

```
$ git cherry-pick -Xtheirs b942f21..d9e8b2c
[master 45bb7e9] C5
 Date: Tue Jun 7 11:00:26 2016 -0400
 1 file changed, 1 insertion(+), 1 deletion(-)
```

It only mentions C5, not C3 and C5. And if you create a log, you only see C5 as cherry-picked.

```
$ git log --oneline
45bb7e9 C5
34142bf C7
de6b1b2 C4
d3b906a C2
b4741c6 C1
```

So, what's the problem? As it turns out, when you specify a range with cherry-pick, the range is interpreted as "everything *after* the starting value and up to and including the ending value." In order to actually include the starting value, you have to tell Git to use the commit that's one before that one. You could just use the SHA1 value for C2 as your starting commit, but to avoid having to look up another SHA1 value if you don't know it, you can just use the caret symbol (^). I previously referred to this as the *caret parent* because adding it to the end of a SHA1 value (or something that resolves to a SHA1 value) means *one before* or *the parent of*.

So, let's fix the command to include C3. First, you reset back one step, before the last cherry-pick. The command is *git reset --hard HEAD^*.

Now, you rerun your cherry-pick command with the desired syntax.

```
$ git cherry-pick -Xtheirs b942f21^..d9e8b2c
```

This time, the output looks more like what you expected.

```
[master 11a6344] C3
 Date: Tue Jun 7 10:59:52 2016 -0400
 1 file changed, 1 insertion(+), 1 deletion(-)
[master 247b261] C5
 Date: Tue Jun 7 11:00:26 2016 -0400
 1 file changed, 1 insertion(+), 1 deletion(-)
```

And if you look at a log of master, you can see what you expected there, too.

```
$ git log --oneline
247b261 C5
11a6344 C3
34142bf C7
de6b1b2 C4
d3b906a C2
b4741c6 C1
```

Differences between Cherry-Pick and Rebase

By this point, you may be thinking that cherry-pick and rebase seem similar in their approach. And, in certain cases, where you are dealing with commits at the ends of branches, the changes that are applied could end up being the same.

In earlier versions of Git, it could be said that cherry-picking was intended more for working with individual commits, but because cherry-pick has been enhanced to accept ranges, that has become less of a distinction.

One key difference, though, is that rebasing changes the base for a branch, which effectively moves a portion of the branch. Cherry-picking selects pieces from a branch, updating the destination branch but not changing or moving the branch the commits originated from.

So, in general, if you need to change the foundation of a branch or update a branch to be based off of another one, you can use rebase. If you need to selectively grab specific commits from another branch and incorporate them into your current branch, you can use cherry-pick.

> **NOTE** *As you may have noticed in the illustrations for the operations, the original commits still exist; the merge and rebase and cherry-picking operations do not move or change the original commits. They use these commits as the source for computing the differences to apply to the destination.*
>
> *This emphasizes something I noted in Chapter 1 on the advantages of Git: Git makes it very hard to "lose" anything.*
>
> *Although you may not have the same pointers (such as a* branch name*) to the original chain of commits, the chain will still exist after the operation. It is really only when garbage collection (git gc) is run manually (or via a preset policy) that commits that are not needed anymore are removed.*

Merge Operations

Throughout the rest of this chapter, I will be talking about operations that can apply to regular merging, rebasing, and cherry-picking. For simplicity, I will refer to these operations by the generic name *merge operations*. This avoids needing to specify *merging and rebasing and cherry-picking* each time.

Undoing Merge Operations

At some point, you are likely to complete a merge operation and then wish you hadn't. Fortunately, if this happens, Git makes it easy to *undo* it and get back to where you were before the merge operation.

There are a couple of aspects to this undo operation. The first one is that for merge operations, the original commits that a branch pointed to before the operation are still there, at least for some period of time. Eventually, Git's garbage collection function may remove them, but Git tries very hard not to let you lose any changes that have been made in the repository. (See the note in the previous section.)

These commits may no longer have a branch pointing to them, but they can still be referenced by their SHA1 values or by an internal Git reference that previously recorded their SHA1 value.

The second aspect that makes an undo possible is the reset command in Git. I talked about this command in Chapter 7 where I used it to reset a local environment (local repository, staging area, and working directory) back in sync with a previous commit.

Combining these two aspects, you can effectively undo a merge operation by resetting your local environment back to the SHA1 value of the commit that was current before you started the merge operation. The command to do so would look something like this:

```
$ git reset --hard <SHA1 value that was current before the merge operation>
```

> **WARNING** *Make sure that you are in the branch that you were merging into or the desired branch before running the reset. If you have multiple branches that have been merged, you may need to do resets in multiple branches with the appropriate SHA1 values for each branch.*

Finding the Right SHA1 Value

The trick here, of course, is finding the right SHA1 value to use to get back to the point before the merge operation. Several options are available.

ORIG_HEAD

I've talked before about the special pointer HEAD that Git maintains to point to the current branch and current commit. Git stores the corresponding reference for whatever HEAD points to as data in the file .git/HEAD. This is generally a reference to another reference for the branch, such as refs/heads/master. If you then look at .git/refs/heads/master, you can actually get the designated SHA1 value for the commit that HEAD (ultimately) references.

When a merge operation happens in Git, Git also saves off another reference named ORIG_HEAD. This value is stored in .git/ORIG_HEAD. It contains the SHA1 value of the commit that was current before the last merge operation. And, like HEAD, it can be used as a reference to Git commands. So, if you haven't done any other merge operations since the merge operation that you want to undo, you can reset back to ORIG_HEAD. Note that the warning I mentioned previously applies here as well: make sure you're in the intended branch that you want to reset before issuing the command.

Here's an example of using the ORIG_HEAD reference. Suppose you have your two branches from the earlier fast-forward merge example: *fix* and *master*. Their logs show the current set of commits in each branch.

```
$ git log --oneline fix    $ git log -oneline master
cb323c7 C5                 31adc9b C3
eb24943 C4                 336715c C2
31adc9b C3                 7058e67 C1
336715c C2
7058e67 C1
```

Now you do the fast-forward merge.

```
$ git checkout master
Switched to branch 'master'

$ git merge fix
Updating 31adc9b..cb323c7
Fast-forward
 file1.txt | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Afterward, you can see the merged results in master.

```
$ git log --oneline master
cb323c7 C5
eb24943 C4
31adc9b C3
336715c C2
7058e67 C1
```

Now, let's look at the contents of HEAD and ORIG_HEAD.

```
$ cat .git/HEAD
ref: refs/heads/master

$ cat .git/ORIG_HEAD
31adc9bd0113af5d48294878597f22e55d863fb0
```

Note that ORIG_HEAD contains the SHA1 value for C3—which was the original HEAD of master prior to the merge operation. So, to get back to the previous state, you can use the reset command to reset back to that point.

It is not strictly required, but you will also use a different option to reset instead of --hard. That option is *--merge*. The difference between --hard and --merge is that --hard discards all changes in the working directory, while --merge does not discard changes that haven't been staged. So, if you have other changes you've started making since the merge, using the --merge option allows you to keep those changes. The command is `git reset --merge ORIG_HEAD`.

And, if you look at the log after this reset, you'll see that master is back to where it was before the merge.

```
31adc9b C3
336715c C2
7058e67 C1
```

Reflog

Reference logs (*reflogs* for short) are additional logs that Git maintains for a reference as it changes over time. I've talked about references a lot throughout this chapter, but I can now define this term more formally. (Reflogs are also covered in the Advanced Topics section of Chapter 7.)

A *reference* in Git terminology refers to a name that you use to refer to a SHA1 value. For example, the default branch in Git is master, and, at any point when using Git, master refers to the current content on the branch. This is the SHA1 value of the most recent commit. You could reference that commit by its SHA1 value or by referencing *master*.

In the Git repository in the .git directory, references are stored in a path structure. At the top is refs, then the type, and then the name. For example, the current SHA1 value for the current commit in the master branch would be stored in *.git/refs/heads/master*. If you are on the master branch, then this should match up with the SHA1 value from the latest change.

```
cat .git/refs/heads/master
373f47835befd4bc24f5b0109eb96a305a15863e
$ git log -1
commit 373f47835befd4bc24f5b0109eb96a305a15863e
Author: Brent Laster <bl2@nclasters.org>
Date:   Tue Mar 29 20:39:20 2016 -0400
remove extraneous files
```

Now, as content is committed into the repository, new SHA1 values become the most current and the SHA1 values in the reference files change. Other activities, such as switching branches, can cause other references to change, such as HEAD, which tracks the current branch.

While there is only one current value for any of these references, a *reflog* for each reference records the values as they change over time. Being able to see how these values change provides another record of what has been done in the system. The reflogs can provide useful information on past points you may want to go back to in the history of the reference. They also record the points where branches are changed—and, most importantly for you right now, the points where merges happened.

References in reflogs are indexed by how many steps back the change occurred. The most recent change for the reference would be at HEAD@{0}. The change before that would be at HEAD@{1}, the one before that at HEAD@{2}, and so on.

The git reflog command is used to look at these changes. The syntax is straightforward: `git reflog <subcommand> <options>`.

I won't dive into more detail on this command right now. I simply want to use it to show the changes for a reference.

Going back to the fast-forward example, if you are on master (because that's where you merged into), you can take a look at the reflog after the merge.

```
$ git reflog
cb323c7 HEAD@{0}: merge fix: Fast-forward
31adc9b HEAD@{1}: checkout: moving from fix to master
cb323c7 HEAD@{2}: commit: C5
eb24943 HEAD@{3}: commit: C4
31adc9b HEAD@{4}: checkout: moving from master to fix
31adc9b HEAD@{5}: commit: C3
336715c HEAD@{6}: commit: C2
7058e67 HEAD@{7}: commit (initial): C1
```

Looking at this, you can see that before the merge, master was at 31adc9b (which corresponds to C3 in the earlier log). You can then reset to that SHA1 value or you can simply use the reflog reference. The example command here would be `git reset --hard HEAD@{1}`.

Afterward, master returns to the place it was before the merge, as do your staging area and working directory. Remember that the relative forms of reflog and reset expect you to be on the branch that you want to work with.

Log

Another simple way to identify the commit to go back to is, of course, to simply look in the *git log* output for the destination branch. Once the correct SHA1 value is identified, the *git reset* can be done using that value.

The trick with this approach is that the user has to be able to figure out which SHA1 value was current before the merge. If the changes are simple and recent enough, this may be the quickest route.

Tag

The tag approach requires some preparation before the merge, but it can greatly simplify things if you think there's a chance that you may want to *undo* the merge later.

The idea is simply to tag the current HEAD before doing the merge with a suitable tag. It can be as simple as *git tag before_merge*. Then, if you change your mind after the merge, you can simply run this command: *git reset --merge before_merge*.

DEALING WITH CONFLICTS

Up until now, the examples in this chapter were based on the assumption that everything merged cleanly and there were no conflicts. This is useful for learning about the merge operations, but not very realistic. Conflicts can arise when doing any of the merge operations, and it is important to understand how Git handles these cases and the workflow you can use to resolve them.

Merging Is a State

One of the first things to understand is that merge operations are states in Git. This means that when the operations are started, Git enters a state that disallows changing contexts until the operation is complete. If everything automatically merges cleanly, then this will likely occur so quickly that you won't notice the change in state. However, if there are conflicts that require user intervention to solve them, you won't be allowed to perform operations like switch branches until you either resolve the conflicts (so the merge operation can complete) or abort the operation.

If your terminal's prompt is configured to show Git information, you see this state in the prompt. Where it normally shows what branch you're working on, in a merge-style operation, it changes to reflect the operation in progress. For example, the prompt will display REBASING or MERGING or CHERRY-PICKING. The following output shows an example.

```
Developer@DESKTOP-80SLL4U MINGW64 ~/cpick (master)
$ git cherry-pick c7a2be5
error: could not apply c7a2be5... C5
```

```
Developer@DESKTOP-80SLL4U MINGW64 ~/cpick (master|CHERRY-PICKING)
$ git checkout feature
file1.txt: needs merge
error: you need to resolve your current index first
```

Note that you are in the CHERRY-PICKING state until you resolve the merge conflict or abort the operation. You are also not allowed to switch branches because you have an unresolved merge in the current branch.

Of course, in the best-case scenario, the merge-style operation completes successfully. There is merged content in the working directory and the local repository is automatically updated.

During a merge, Git processes all of the files that it can cleanly merge and resolve. If it encounters files with conflicts that it cannot resolve, it stops (still in that state), and waits for you to resolve the conflicts and stage the fixes. Before I discuss how to proceed, let's take a quick look at how Git informs you that there are conflicts for the different merge operations.

Error Messages for Conflicts

Because the different merge operations have different behaviors, there are different error messages for each operation when conflicts occur. Depending on the operation, you may have more or less information and guidance to work with. I include some examples here to illustrate the different forms. (In the following section, I will discuss how to resolve these conflicts along the lines of what's suggested in the messages.)

For the actual merge command, you may see something like this:

```
$ git merge feature
Auto-merging file1.txt
CONFLICT (content): Merge conflict in file1.txt
Automatic merge failed; fix conflicts and then commit the result.
```

For a rebase, it's a bit more complicated:

```
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: C3
Using index info to reconstruct a base tree...
M       file1.txt
.git/rebase-apply/patch:8: trailing whitespace.
"C3"
warning: 1 line adds whitespace errors.
Falling back to patching base and 3-way merge...
Auto-merging file1.txt
CONFLICT (content): Merge conflict in file1.txt
error: Failed to merge in the changes.
Patch failed at 0001 C3
The copy of the patch that failed is found in: .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".
```

If you scan the output carefully, you can find the same conflict message as you saw in the merge case. However, notice there is a lot of additional information, and you have options such as *--continue* and *--skip*.

To understand this, think back to the example I used earlier in this chapter to show how the rebase process works. Rebase typically involves taking the deltas of multiple commits to reproduce a history of multiple changes on another branch. So, because there are usually multiple commits involved, each one is a separate *merge* operation, done in a sequence.

The message is telling you that it tried to apply the delta of one of the commits involved in the rebase and it got a merge conflict. Now it's up to you to resolve that conflict if you want, and then tell it to *continue* (which means try to apply the delta for the next commit from the history) or just *skip* this one and let it continue. Or, you can *abort* the whole rebase operation (which I'll talk about in a moment).

Notice also that Git tells you where you can see what it was trying to do—in the rebase-apply/patch file in the .git directory. If you were to look at that file, you would probably see something like this:

```
$ cat .git/rebase-apply/patch
diff --git a/file1.txt b/file1.txt
index 65f00c63fbf892d06956fcbe9b3a5895db7fecbf..96a36f68be95938938e2367347e9d68dd0a
e4d67 100644
--- a/file1.txt
+++ b/file1.txt
@@ -1,2 +1 @@
-C1
-"C2"
+"C3"
```

For a cherry-pick operation, you get a message about the conflict as well, though in yet another format.

```
$ git cherry-pick c7a2be5
error: could not apply c7a2be5... C5
hint: after resolving the conflicts, mark the corrected paths
hint: with 'git add <paths>' or 'git rm <paths>'
hint: and commit the result with 'git commit'
```

Aborting the Operation

When you encounter one of these conflict situations, you can resolve the conflicts, stage and commit the results, and finish the operation. However, you also have another choice.

If you decide, for whatever reason, that it is not worth resolving the conflicts and completing the merge, you can abort the merge-style operation to get out of that state. This returns your local environment to the way it was before you ran the merge-style command.

To abort one of these operations, you just run the merge-style command with the --abort option as follows:

```
$ git merge --abort
$ git rebase --abort
$ git cherry-pick --abort
```

Dealing with Conflicts—the Workflow

In the earlier examples, I illustrated the changes in the repository when merge operations were done. I didn't focus on the working directory or staging area as part of the process because, if everything merges cleanly, Git just takes care of updating everything for you in the working directory and local repository.

However, where conflicts occur, the working directory, staging area, and local repository all have key roles to play. As well, the user must interact with all of them to resolve the conflicts and complete the merge.

In a case where conflicts occur, Git takes the following actions:

- If any files merged successfully, Git adds them to the staging area (stages them).
- For the files that have conflicts, Git inserts conflict markers in the files and leaves the versions with the markers in the working directory.
- Git alerts the user (refer back to the section "Error Messages for Conflicts").

To help you understand the point of this workflow, I note again that if a merge operation has no conflicts, the files are merged in the working directory and end up in the local repository with no user intervention. It's as if Git staged and committed the entire set of merged content for you. However, when there are conflicts that Git can't resolve, it can't do that entire process. So, it does as much as it can.

For the files that merged cleanly or didn't need to merge, Git goes ahead and stages them. This involves using the staging area as the *Prepare* case I talked about in Chapter 3. Because the commit you are trying to merge is treated as a unit, Git wants to have all of the updated files staged so they can then be committed as a unit to complete the merge. Git does what it can toward this goal by staging the files that don't have unresolved conflicts.

For files that have conflicts that Git can't resolve, it puts them in the working directory with the merge conflicts marked. If you're working in the command line, the conflicts will be marked with <<<< and >>>> lines. (See the section "Alternative Style for Conflict Markers," for more details.)

The idea is that the user will resolve all the remaining conflicts in whatever way is appropriate, and then will stage all of those files. Those files, in combination with the files that Git staged previously, comprise a complete set in the staging area.

The last step is to merge that complete set that makes up the commit into the local repository and thus complete the merge. You end up with the complete set merged in the working directory and in the local repository, just as you would if everything had merged cleanly in the first place.

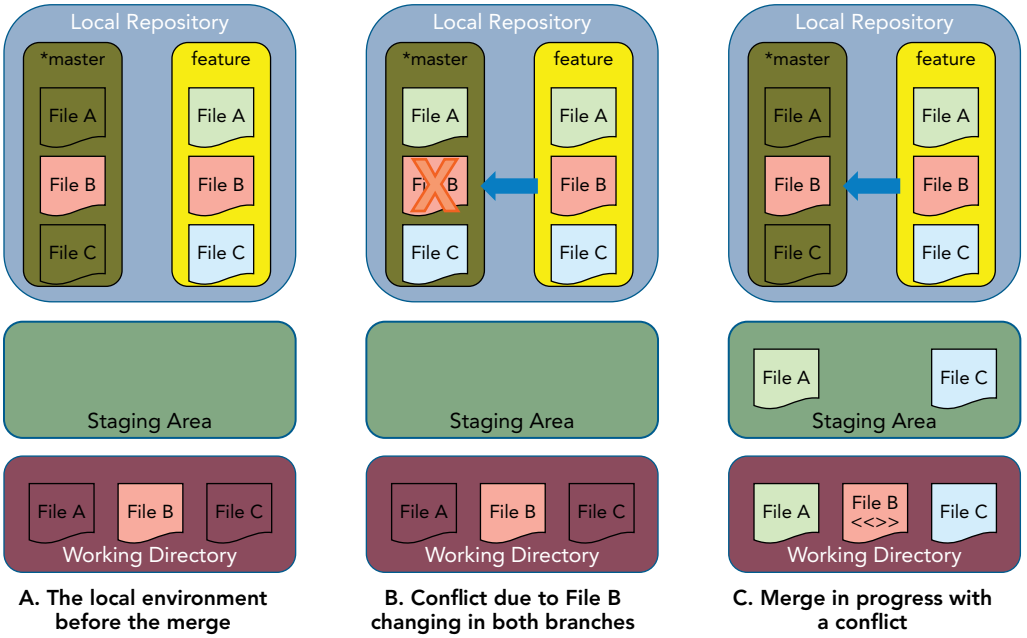
Once the merge is completed by resolving the conflicts, staging the fixed files, and doing the commit of the full set, Git is happy and the MERGING state is ended.

Visualizing the Merging Workflow with Conflicts

The parts of Figure 9-14 illustrate the merge process. Here, you have a couple of branches in the repository that you want to merge together—*feature* into *master*. I'll use these pictures to help you

visualize the process. (Note that I will not show all of the detail here—just illustrations for how to think about merging.)

The setup in Figure 9-14, part A, is that *all three* files have been changed in *feature*, and *only* File B has been changed in *master* (as indicated by the shading patterns). *Master* is your current branch (as suggested by the *) and you have the files from *master* in the working directory.



A. The local environment before the merge

B. Conflict due to File B changing in both branches

C. Merge in progress with a conflict

FIGURE 9-14: The merge process in the local environment

Now, you can start the merge operation.

```
$ git checkout master
$ git merge feature
```

Figure 9-14, part B, represents the merge process in the repository. As Git tries to merge *feature* into *master*, there is a conflict (as suggested by the X) because File B has been changed in both branches.

In cases like this, Git follows the process I discussed earlier. It stages the files that merged successfully (puts them into the staging area). For any files that have conflicts, those conflicts are marked with the Git conflict markers (<<<<<<<<<< and >>>>>>>>>>). These files are then put into the working directory so the conflicts can be resolved by the user. This state is illustrated in Figure 9-14, part C.

Now, it is up to the user to resolve the conflicts that Git couldn't merge, and complete the operation. At this point, if you are using a terminal configured to show Git information in the prompt, the prompt will change to (MERGING...), meaning you are locked into that state until you resolve the operation by completing the merge or abandoning it.

Which Files?

The central idea here is that Git wants to have a clean set of content (merged files) to stage and then commit to finalize the merge. You are trying to assemble a full set of content from the snapshot in the staging area to commit as a merge result.

When some files are successfully merged and others have conflicts, the immediate question becomes, "Which files merged cleanly and which did not?" You need to know this so that you know which ones to fix. An easy way to get this information is with the *git status* command.

In the current example, at this point, you have two files, File A and File C, that merged cleanly and so were staged. File B was modified on both branches (*master* and *feature*) and has conflicts that have to be resolved manually. If you execute *git status* in this environment, you see the following message:

```
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

Changes to be committed:

        modified:   File A
        modified:   File C

Unmerged paths:
  (use "git add <file>..." to mark resolution)

        both modified:   File B
```

If you recall the information on git status from Chapter 6, this message should look familiar. Let's break it down.

- **Changes to be committed:** This implies that these files are in the staging area because the next destination is the local repository where they are *to be committed*.
- **Unmerged paths:** This refers to files that could not be merged for some reason and need manual intervention.
- **Both modified:** This refers to files modified on both branches.

The status information from Git points out which files were processed cleanly from the merge and which ones still need someone to look at them. Using this information, you know which files you need to look at.

Role of the Staging Area

The files that merged successfully are staged, and there is nothing you need to do with them. You might be tempted to try and commit them. However, at this stage, the merge is not finished, and because it is still in progress, Git is locked in the merging state. So, you need to look at how to resolve the conflicts.

NOTE *Besides the conflict markers in the local file, there are other ways to look at the diffs when you have a merge conflict. One way is to use the command* git log --merge -p <path>.

When you use this command, Git shows you the diffs for the two versions, in sequence, against the common ancestor. For example, the following output shows the differences for a simple file, file1.txt, with the original version with contents master, *that has been updated for both the master branch and a feature branch, and now has merge conflicts.*

```
$ git log --merge -p file1.txt
commit 75092c13bf8e99142a6c3b71079e2cf7a21bb62e
Author: Brent Laster <bcl@nclasters.org>
Date:   Tue Jun 7 23:06:22 2016 -0400

update on master

diff --git a/file1.txt b/file1.txt
index 1f7391f..9e99285 100644
--- a/file1.txt
+++ b/file1.txt
@@ -1 +1 @@
-master
+update on master

commit 59f04da402d04dd7b42b96d7dab47a37596c81a4
Author: Brent Laster <bcl@nclasters.org>
Date:   Tue Jun 7 22:32:17 2016 -0400
from feature

diff --git a/file1.txt b/file1.txt
index 1f7391f..a7453f0 100644
--- a/file1.txt
+++ b/file1.txt
@@ -1 +1 @@
-master
+feature
```

You can also use the git show *command to see the different versions. If you pass* :#:filepath *as the arguments, then you can see the various versions involved in the merge. Using 1 for the # in the expression shows the common ancestor; using 2 for the # shows the tip of the current branch; and using 3 for the # shows the result after the merge was attempted.*

```
$ git show :1:file1.txt
master
$ git show :2:file1.txt
still master
$ git show :3:file1.txt
feature
```

To resolve the conflicts and finish the merge, you need to know which files are in conflict. The *git status* command, as I described here, is the easiest way to get such a list. You can also employ *git diff* to show differences.

The conflicts can then be resolved through editing or via a graphical merge tool (discussed in the section "Visual Merging"). Alternatively, the merge can be aborted and then rerun with a different merge strategy that may further resolve some conflicts. (I discuss merge strategies in more detail in the next section.)

After the conflicts have been resolved, the edited files in the working directory need to be staged into the staging area. This is so that you will have a complete set of files ready to be committed. Doing this is simply a matter of running the command *git add [. or individual files]*.

Completing the Merge Operation

Prior to staging the updated files that are identified as conflicts, Git does not allow you to just do a commit of what is in the staging area to finish the merge. The reason again is that Git wants a full set of merged content to complete the operation. Git doesn't want partially merged branches in its repositories. (For this reason, you are not allowed to use the *-am* shortcut on the commit command, when completing merge, to bypass the staging area.)

However, once you have the full set of files in the staging area (those that originally merged cleanly and were staged by the merge operation, plus the ones that didn't merge cleanly but you fixed and staged), you can commit the entire set as an update. An example would be *git commit -m "Finalize merge"*.

This promotes the fully merged commit into the local repository. At that point, your merge is complete and Git takes you out of the *merging* state, completing the workflow. You are once again able to do normal operations.

Resolution Options and Strategies

Git supports a variety of merge strategies and options for the merge operations. These tell Git how to attempt to merge items and how to resolve conflicts. You can select the strategy you want to use via the *-s* option, and the *–X* option allows you to pass arguments to the chosen strategy. I briefly describe the different strategies in the following sections.

Resolve Strategy

The resolve strategy is a limited strategy that can only merge two things (branches) using a three-way merge. However, being simpler, it may be faster in some cases.

Recursive Strategy

Recursive is the default strategy when you're merging two branches together. Here, the recursive nature comes into play if there is more than one common ancestor for the two branches. In that case, it does a merge of the common ancestors and then uses that merge as a basis for the three-way merge.

The recursive strategy takes a number of options. These are passed with the -X option to any of the merge operations. You'll look at those options next.

Recursive Strategy Options

The recursive strategy includes a number of useful options. I'll note a few of the most useful ones here; others can be found on the help page for *git merge*. In situations where you're merging two things together, because recursive is the default strategy, you can just pass the -X<option> without including -s to specify the strategy.

- **Ours:** This option tells Git that when a file has been modified on both branches, resulting in a conflict, it must use the version from the current (destination) branch as the resolution.
- **Theirs:** This option tells Git that when a file has been modified on both branches, resulting in a conflict, it must use the version from the source branch as the resolution.

Note that these options only apply in the case of conflicts. If a file has been changed on one or the other branch so that there is no conflict, the normal merge behavior applies.

These options tell Git to ignore the indicated whitespace changes.

- ignore-space-change
- ignore-all-space
- ignore-space-at-eol

Recursive Strategy Example

Let's look at a quick example of the recursive strategy. I'll use some captures from the gitk tool to illustrate them. Suppose you have a master branch and three other branches off of that branch, as shown in Figure 9-15.

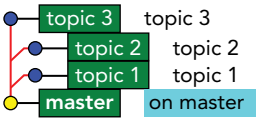


FIGURE 9-15: Master branch with three topic branches

If you merge in the topic branches one at a time, Git attempts to first do a fast-forward merge. If it can do a fast-forward merge, it does (unless you supply the --no-ff option). If it can't do a fast-forward merge, because you are merging two branches at a time, it defaults to the recursive strategy.

```
$ git merge topic1
Updating
0d89cc5..59d9e1a
Fast-forward
 topic1.txt | 1 +
 1 file changed, 1
insertion(+)
 create mode 100644
topic1.txt
```

```
$ git merge topic2
Merge made by the
'recursive' strategy.
 topic2.txt | 1 +
 1 file changed, 1
insertion(+)
 create mode 100644
topic2.txt
```

```
$ git merge topic3
Merge made by the
'recursive' strategy.
 topic3.txt | 1 +
 1 file changed, 1
insertion(+)
 create mode 100644
topic3.txt
```

Afterward, your branch arrangement looks like Figure 9-16. To understand the arrangement, look at it from bottom to top and right to left. The lines to the right of the dots represent where you started. The *topic1* branch was fast-forwarded into *master*. Then the *topic2* branch was merged in using the recursive strategy, and likewise for *topic3* to arrive at the new master (at the top).

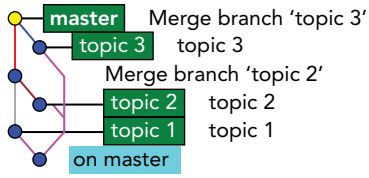


FIGURE 9-16: After a merge of the three topic branches

Finally, here's an example of how the recursive strategy options can be useful. Suppose you have the branch setup from the earlier cherry-pick example, as shown in Figure 9-17.

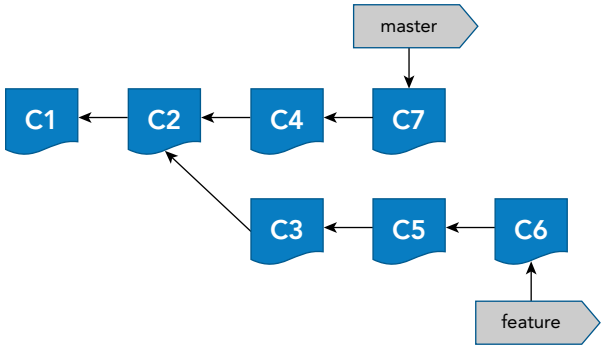


FIGURE 9-17: The earlier cherry-pick example

You attempt to cherry-pick C5 (using its SHA1 value) from *feature* onto *master*, but this time you encounter a conflict because C5 cannot be applied cleanly to *master* (some common element in it has been changed on both branches since the common ancestor back at C2).

```
$ git cherry-pick d9e8b2c
error: could not apply d9e8b2c... C5
hint: after resolving the conflicts, mark the corrected paths
hint: with 'git add <paths>' or 'git rm <paths>'
hint: and commit the result with 'git commit'
```

This is illustrated in Figure 9-18. If you now need to see which files are causing the conflicts, you can use the *git status* command. Here, I show both the long version and the short version.

```
$ git status
On branch master
You are currently cherry-picking commit d9e8b2c.
  (fix conflicts and run "git cherry-pick --continue")
```

```
  (use "git cherry-pick --abort" to cancel the cherry-pick operation)

Unmerged paths:
  (use "git add <file>..." to mark resolution)

        both modified:   file1.txt

no changes added to commit (use "git add" and/or "git commit -a")

$ git status -sb
## master
UU file1.txt
```

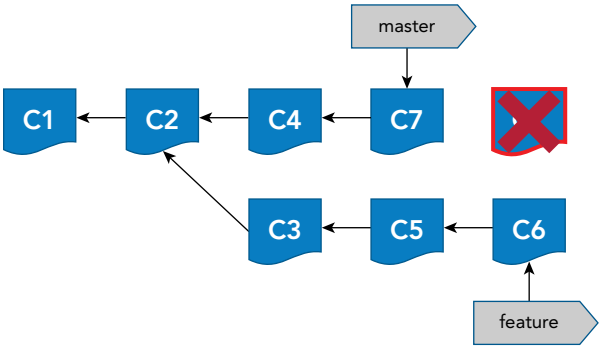


FIGURE 9-18: C5 cannot be cherry-picked due to a conflict.

You are now in the *cherry-picking* state until you resolve or abort the operation. Suppose in this case that you decide you just want the version of C5 from *feature*. You can use the options to the recursive strategy to do this. However, you first need to figure out whether you need *ours* or *theirs*. Recall that in these cases, *ours* refers to your current branch (the one you are merging into, or the destination) and *theirs* refers to the source branch (the one you are merging from). Figure 9-19 highlights the choices as they would be used for options to the recursive strategy.

So, using this information, you can abort the current operation and rerun your cherry-pick with the additional option to specify the version you want (and avoid the conflict). The command would be *git cherry-pick -Xtheirs d9e8b2c*.

And the output shows that it worked.

```
[master 9f308b6] C5
 Date: Tue Jun 7 11:00:26 2016 -0400
 1 file changed, 1 insertion(+), 1 deletion(-)
```

This leaves you with the structure shown in Figure 9-20.

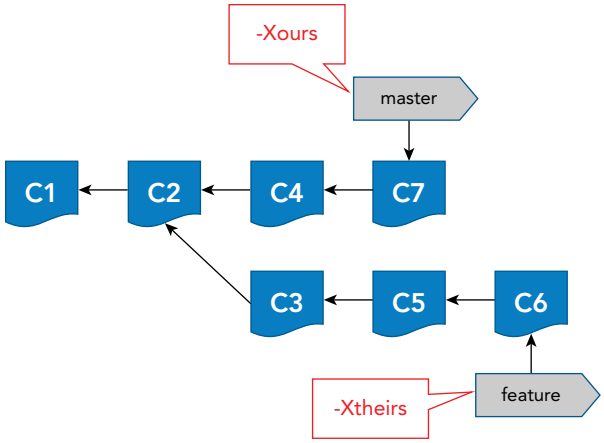


FIGURE 9-19: The choices for options to pick one version

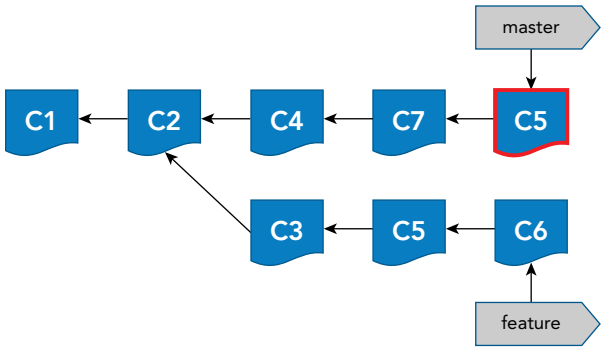


FIGURE 9-20: Completed cherry-pick with C5 from feature

Octopus Strategy

Presumably, the name of the octopus strategy derives from the idea of being able to handle a large number of branches like the arms of an octopus. This strategy is used when merging in more than one branch to the current one. It is the default strategy in that case.

As an example of using the octopus strategy, you'll look at the same arrangement I used in the recursive strategy example. Refer back to figure 9-15 for the gitk presentation of master and the three topic branches.

You can actually tell Git to merge in all three branches in the same command. When you do this, Git uses the octopus strategy instead of the recursive one. In this example, you specify the --no-ff

(no fast forward) flag to make the end result clearer. Git still suggests that it is doing a fast-forward merge, but it actually doesn't because of the flag.

```
$ git merge --no-ff topic1 topic2 topic3
Fast-forwarding to: topic1
Trying simple merge with topic2
Trying simple merge with topic3
Merge made by the 'octopus' strategy.
 topic1.txt | 1 +
 topic2.txt | 1 +
 topic3.txt | 1 +
 3 files changed, 3 insertions(+)
 create mode 100644 topic1.txt
 create mode 100644 topic2.txt
 create mode 100644 topic3.txt
```

Notice that although you don't supply a strategy, Git defaults to the octopus strategy in this case. After this, your branch layout looks like Figure 9-21. (Again, read the changes from bottom to top and right to left.)

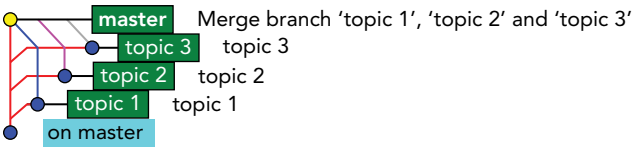


FIGURE 9-21: After the octopus merge

Ours Strategy

There is also a merge strategy named *ours*. It is not particularly useful, but I mention it here to differentiate it from the option of the same name for the recursive strategy.

The difference here between the ours strategy (-s ours) and the ours option to the recursive strategy (-X ours) is this: using the strategy tells Git to keep everything from the current branch and not to overwrite anything. Using the option tells Git to just keep your current content if there's a conflict. When using the options with the recursive strategy (-X), it's possible to still get content from the other branch if there's no conflict. With the strategy (-s), nothing gets overwritten.

Strategies—Conclusion

Aside from manually fixing conflicts yourself, the ours and theirs options will probably handle most cases where you need to merge two branches in particular ways. For merging in more than two branches, you have a choice of doing them two at a time with the recursive strategy and the options it makes available, or just letting the octopus strategy do what it can.

Note that there are additional options available for the recursive strategy and several other strategies for specialized cases. Details on those strategies can be found on the git merge help page

(*git merge --help*). That is the central source for information about merging, regardless of which operation you are doing.

VISUAL MERGING

As you have seen, the default presentation for merge conflicts is to insert a series of "<" and ">" signs around the conflicts indicating which branch they originated in. You can also use the diff command to show the conflicting changes in a standard patch format. While I am focusing primarily on command line usage throughout this book, as I noted in the section on diffing, there are times when a visual interface adds significant value or convenience. Merging is another one of these cases.

Git includes a special command for working visually with differences: *mergetool*. The syntax for this command is as follows:

```
git mergetool [--tool=<tool>] [-y | --[no-]prompt] [<file>...]
```

You would normally run this command after doing a *git merge* and after Git has identified that there are conflicts that need to be resolved manually. If you run this command and supply one or more files for the command to work with, Git runs the tool multiple times in succession to handle the merging for each file you specify. (Note that if you specify files that don't need a merge, they are skipped.) Specifying a directory limits the tool to all files with unresolved conflicts in that path; running it with no files or directories addresses all files with unresolved conflicts.

The idea is that you have one or more visual merging applications installed, configured, and available for Git to use. (More about how that works in a moment.) Then, you use the git mergetool command line or a configuration value to select the one you want to use. The mergetool command then starts up the desired tool with the appropriate arguments.

Figures 9-22 to 9-25 show some screenshots of several commonly used visual merging applications (and applications that Git understands "out of the box" if they are installed and in the path). If you have read Chapter 6, you will notice that you can use the same tools for resolving merge conflicts visually that you use to see differences visually.

When using these tools to resolve merge conflicts as opposed to just viewing differences, there are typically a couple of differences in the user interface:

- An additional window may be present for showing the base version (the version that was current prior to the merge operation) as well as the two versions being merged together.
- An additional window may be present for showing the results of the actions taken in the merge tool—the resolved version of the file.
- Additional menus and commands are available to assist in choosing content and creating the resolved version.
- In most cases, these tools include clickable or draggable highlights to move selected content between windows. This allows you to use a mouse or similar mechanism to easily select the final content to include in the resolved file.

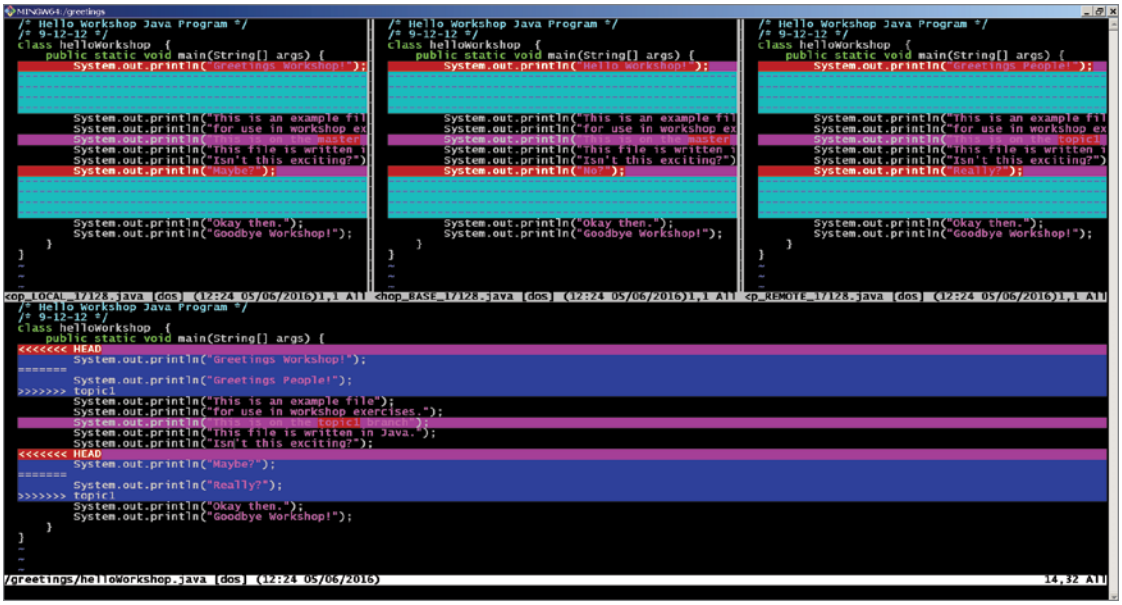


FIGURE 9-22: Merging with vimdiff

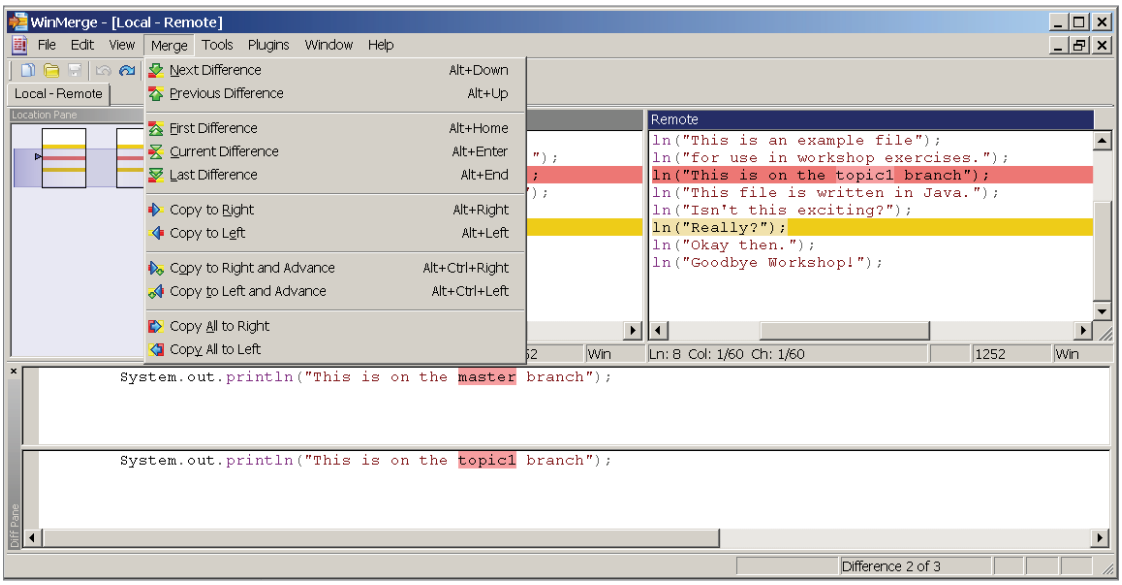


FIGURE 9-23: Merging with WinMerge

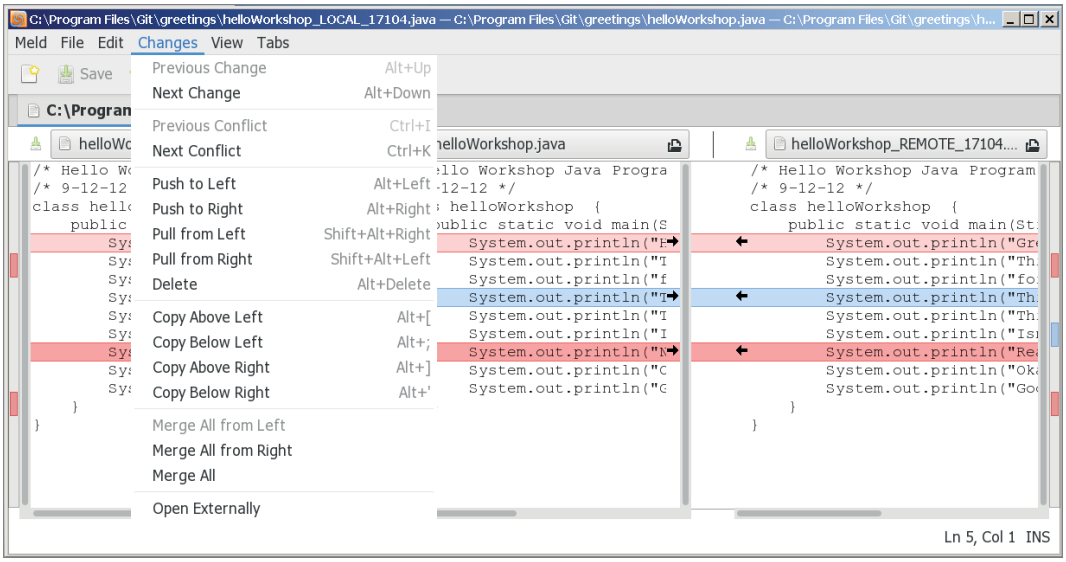


FIGURE 9-24: Merging with Meld

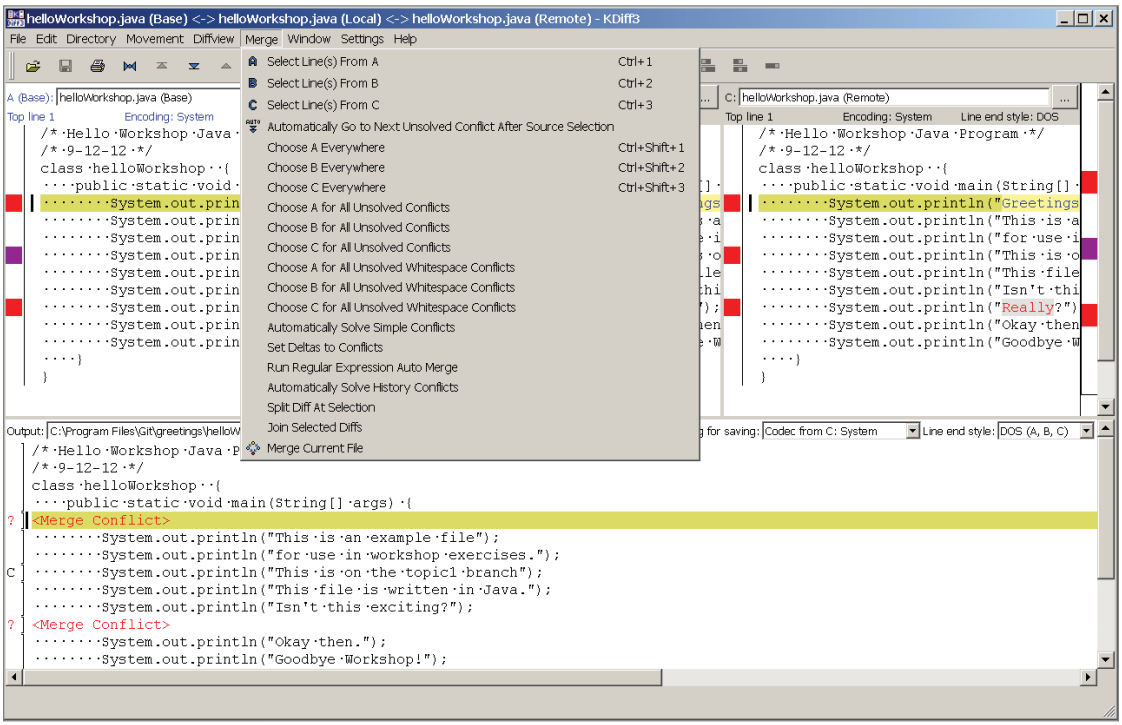


FIGURE 9-25: Merging with KDiff3

Selecting a Merging Tool

While there are a variety of merge tools to choose from, you will probably want to pick one as a default. From a list of installed, configured, and available tools, a particular tool can be selected in several different ways. For the mergetool command, you can use the default or specify just the simple name of the tool (such as kdiff3, vimdiff, or meld).

If no default tool has been specified, then Git attempts to use a sensible default (usually something like vimdiff on Linux systems). One way to configure a particular tool is via the merge.tool configuration value: *git config --global merge.tool vimdiff*. Once this is configured, running *git mergetool* starts that selected tool—vimdiff.

By default, if no tool is specified or configured, Git prompts for confirmation to run the merge tool for each file to be merged.

```
Merging:
helloWorkshop.java

Normal merge conflict for 'helloWorkshop.java':
  {local}: modified file
  {remote}: modified file
Hit return to start merge resolution tool (kdiff3):
```

You can suppress this prompt by supplying a *no-prompt* option when running mergetool, as in *git mergetool --no-prompt*.

Another way to select a particular tool is to specify the name of the desired tool via the -t option when you run mergetool, as in *git mergetool -t meld*.

Making Merge Tools Available to Git

Git comes preconfigured to be able to work with a number of different tools for merging. To see a list, you run the command *git mergetool --tool-help*.

Note that this does *not* mean that all of these tools are installed (or even if installed, that they can be used; they might not be in the path). What this does mean is that Git understands how to use these tools to do merging without additional configuration, if the tool is available on the system. The tool-help option tells you which tools are available to use (under *may be set to the following*) and which are not (under *The following tools are valid, but not currently available*).

To make one of the tools available that is marked as *not currently available*, you can install the application and make sure it is in the path. Once that's done, if it is a tool that Git knows about, it will show up in the *available* section.

If a tool is not available in the path, then you can set a configuration value named *mergetool.<tool>.path* (where <tool> is the name of the application) to specify the location where Git can find it.

For example, Git knows how to work with an application named *meld* for merging when it can find it on the system. Suppose you install the Meld application on Windows in c:\meld (instead of the default Program Files location that would be in the path). To tell Git where the Meld application is found, you can set the path value for it as follows.

On a Windows command prompt:

```
$ git config --global mergetool.meld.path c:\meld\Meld.exe
```

On a Bash shell:

```
$ git config --global mergetool.meld.path /c/meld/Meld.exe
```

> **NOTE** *The configuration of the mergetool.<tool>.path value can also be used to work around differences (such as capitalization) between the actual application name and what Git expects out of the box.*
>
> *For example, Git expects Meld to be* meld *(lowercase). If Meld.exe is installed in a place where Git should be able to see it, Git may still not see it due to the difference in case. When you configure the tool name in mergetool.meld.path, Git can then understand how to find the* meld *tool.*

ADVANCED TOPICS

In this section, you'll take a look at three merge-related items that you may find useful. You'll first look at an alternate way to display conflict information that adds an additional, useful piece of data. Then you'll look at an advanced scenario for rebasing that allows you to filter what you rebase based on intersections with another branch. Finally, you'll see how to use an option of rebase that really is its own feature: interactive rebasing. This unique functionality allows you to script changes to commits in the repository's history.

Alternative Style for Conflict Markers

In a conflict situation, Git marks conflicts in files with strings of "<" and ">" symbols. A string of "<<<<<<<" is used to note the revision in the current branch, and a string of ">>>>>>>" is used to note the other revision from the branch being merged in.

For example, suppose you have a file with a quiz problem in master. You stage and commit that file. Then you create a branch for a guess. You update the file in *master* with one guess and commit that change. You switch to the *guess* branch and update that version with a second guess. So, now your original version has been modified on both branches. The sequence might look something like this:

```
$ echo "c = a*2+(b-3)" > quiz.txt

$ git add .

$ git commit -m "problem"
[master (root-commit) 3288fb4] problem
 1 file changed, 1 insertion(+)
 create mode 100644 quiz.txt

$ echo "if a = 2 and b = 5 then c = 4" > quiz.txt
```

```
$ git branch guess

$ git commit -am "answer 1"
[master a7eee54] answer 1
 1 file changed, 1 insertion(+), 1 deletion(-)

$ git checkout guess
Switched to branch 'guess'

$ echo "if a = 2 and b = 5 then c = 6" > quiz.txt

$ git commit -am "answer 2"
[guess ac1d0b9] answer 2
 1 file changed, 1 insertion(+), 1 deletion(-)
```

When it comes time to pick an answer, you can attempt to merge *guess* into *master*. As you would expect, because both branches have changed the same line, you now have a merge conflict that has to be resolved.

```
$ git merge guess
Auto-merging quiz.txt
CONFLICT (content): Merge conflict in quiz.txt
Automatic merge failed; fix conflicts and then commit the result.
```

Taking a look at the file with conflicts, you can see the two changes.

```
$ cat quiz.txt
<<<<<<< HEAD
if a = 2 and b = 5 then c = 4
=======
if a = 2 and b = 5 then c = 6
>>>>>>> guess
```

This is useful, but one piece is missing here: remembering what the quiz problem was in the original version of the file (the common ancestor). Without that information, it is difficult to know which answer is right.

Luckily, Git provides an alternate way to show conflict information, one that includes the original version (common ancestor) of the changes. You can enable showing the additional information by setting the Git configuration value *merge.conflictstyle* to *diff3*. The full command is *git config --global merge.conflictstyle diff3*. You can set that value, abort the merge, and then rerun the merge operation.

```
$ git merge --abort

$ git merge guess

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

Now, if you look at the file with conflicts, you see something different.

```
$ cat quiz.txt
<<<<<<< HEAD
```

```
if a = 2 and b = 5 then c = 4
||||||| merged common ancestors
c = a*2+(b-3)
=======
if a = 2 and b = 5 then c = 6
>>>>>>> guess
```

Notice the line under the ||||||| *merged common ancestors* line. This is the original version. The diff3 style shows all three pieces: the common ancestor and the two changed revisions. Now you can determine which answer is right.

Advanced Rebasing Scenario

In addition to the *basic* rebase functionality that I discussed in the main part of this chapter, there is more advanced functionality that can be used with rebasing.

The first form is a variation on the previous rebasing functionality I discussed. The difference is that it allows filtering what is rebased against another branch. This allows for very precise *slicing and dicing* of branches during rebase operations.

The advanced syntax looks like this:

```
$ git rebase --onto newbase branch2 [ branch1]
```

The way you interpret this is *Rebase branch1 off of newbase, but exclude any commits that branch1 and branch2 have in common*. So, only rebase the part of branch1 that is *beyond* branch2.

Let's look at an example. In Figure 9-26, you have three branches*: master, feature,* and *topic*. *Feature* is branched off of *master* at C2 and *topic* is branched off of *feature* at C4.

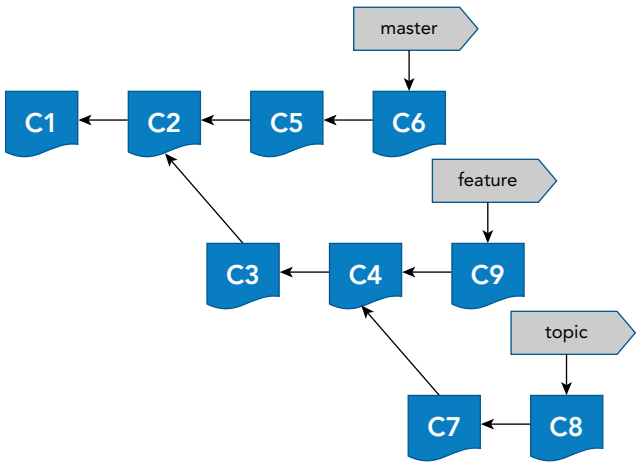


FIGURE 9-26: Setup for an advanced rebase

Now suppose you want to merge everything that is in *topic*, but not in *feature*, onto *master*. As highlighted in Figure 9-27, *topic's* chain includes C8, C7, C4, C3, C2, and C1.

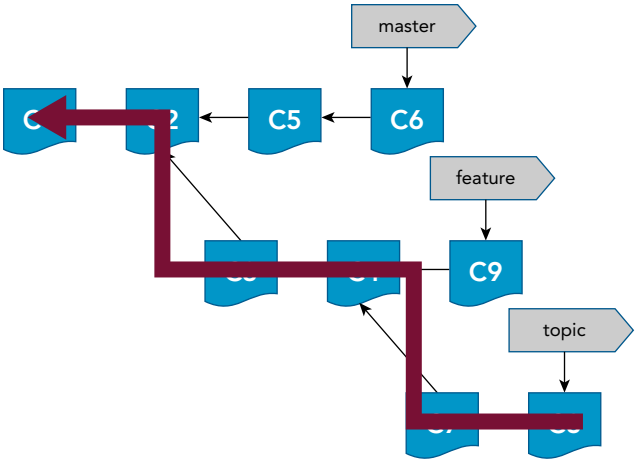


FIGURE 9-27: Topic's chain of commits

A quick Git log bears this out. (Note that I have used the commit names in the figures as comments here to make this clear.)

```
$ git log --oneline topic
7550c8f C8
b83a885 C7
54d5770 C4
6b77391 C3
27cd1a1 C2
ef77f69 C1
```

So, if you issue the command for the advanced rebase as *git rebase --onto master feature topic*, then this tells Git to rebase the parts of *topic* that are not shared with *feature* onto *master*. Figure 9-28 illustrates the logic in determining the set of commits to rebase. Everything from C4 back to C1 is excluded because it is part of *feature*. That leaves C7 and C8. So Git computes the deltas (differences as indicated by the yellow triangles) on those commits.

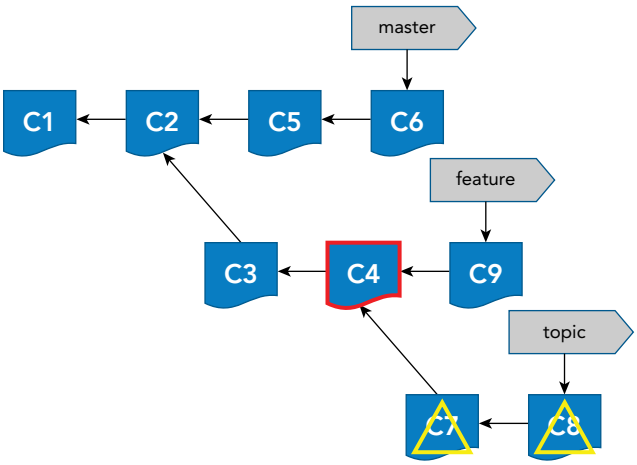


FIGURE 9-28: Computing the deltas to rebase

You can also see the set of commits that are in *topic* but not in *feature* by using one of the advanced forms of the log command.

```
$ git log --oneline topic ^feature
7550c8f C8
b83a885 C7
```

This can be read as "show me everything in *topic* that is not in *feature*." The caret (^) on the front of *feature* is interpreted as *not*.

Now let's get down to actually running the rebase. You have thought ahead and know that you want the commits from the *topic* branch to be what you end up with in the *master* branch. So, to avoid merge conflicts, you add the *-X theirs* option. This tells Git that, in the event of a merge conflict, it must use the version from the branch being merged from.

NOTE *As I discussed earlier, if you omitted the -X theirs option, and you run into merge conflicts, Git stops on the first conflict, and waits for you to resolve it. It might look something like this:*

```
$ git rebase --onto master feature topic
First, rewinding head to replay your work on top of it...
Applying: C7
Using index info to reconstruct a base tree...
M       file1.txt
Falling back to patching base and 3-way merge...
Auto-merging file1.txt
CONFLICT (content): Merge conflict in file1.txt
error: Failed to merge in the changes.
Patch failed at 0001 C7
The copy of the patch that failed is found in: .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".
```

At this point, you have a choice:

- *Resolve this conflict, then run the* git rebase --continue *command to move on to the next conflict (assuming a next one),*

OR

- *Skip this patch via the* git rebase --skip *option,*

OR

- *Abort the rebase with* git rebase --abort *and rerun it with a merge strategy that tells Git how to decide which version to use if there's a conflict.*

It's also worth noting here that if the merge for a particular commit during a rebase would result in the same content that already exists in the target branch, then the commit is effectively skipped.

The command to do this is *git rebase -X theirs --onto master feature topic*.

Based on this command, Git changes to the *master* branch and then computes the deltas for the commits that are on the *topic* branch that are *not* also on *feature*. Figure 9-29 illustrates this. C4 marks the common ancestor between *feature* and *topic* (everything *before* and including C4 is in both *feature* and *topic*). So Git computes the deltas from that point for C7 and C8.

Once the deltas are computed, as in any rebase, Git attempts to replay (apply) those differences as new commits at the end of the target branch (in this case, the branch specified by the --onto argument), *master*. Figure 9-29 illustrates this process.

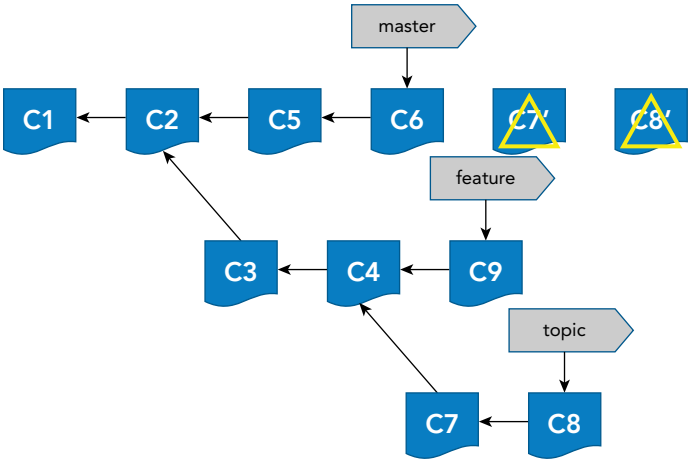


FIGURE 9-29: Applying the deltas to master

If the deltas apply cleanly (or after conflicts have been resolved if you didn't specify the merge strategy), you end up with the unique parts of *topic* now rebased on *master* (see Figure 9-30).

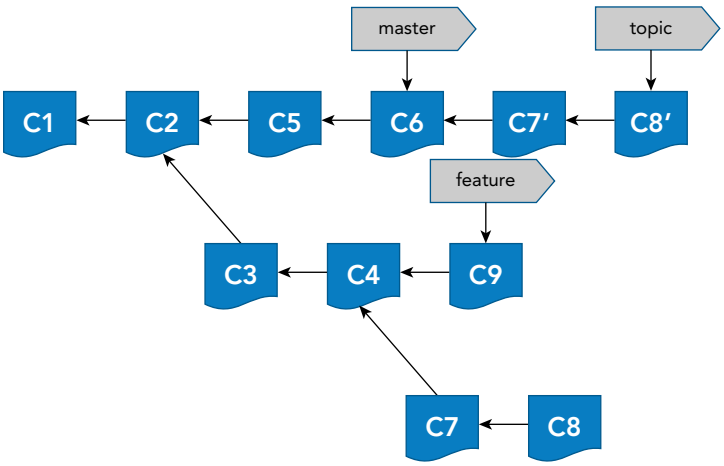


FIGURE 9-30: The completed rebase

Looking at a log of the *topic* branch, you can see that it now includes the unique pieces of the original *topic* branch plus the commits of the *master* branch. And we can also see that master has not been changed; it still points back to C6.

```
$ git log --oneline topic
91897da C8
fd15463 C7
f63b391 C6
a55d7ef C5
27cd1a1 C2
ef77f69 C1
```

```
$ git log --oneline master
f63b391 C6
a55d7ef C5
27cd1a1 C2
ef77f69 C1
```

You can now merge *topic* into *master* by doing a simple fast-forward merge.

```
$ git checkout master
$ git merge topic

Updating f63b391..91897da
Fast-forward
 file1.txt | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

$ git log --oneline master
91897da C8
fd15463 C7
f63b391 C6
a55d7ef C5
27cd1a1 C2
ef77f69 C1
```

The results in the arrangement are shown in Figure 9-31.

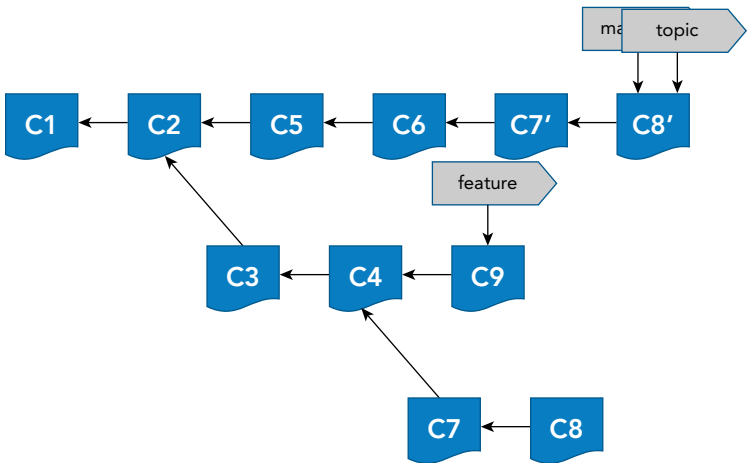


FIGURE 9-31: Topic merged into master

Interactive Rebasing

One more form of rebasing is worth discussing here: *interactive rebasing*. Interactive rebasing provides a powerful way to modify commits that you've already made in the local repository.

In essence, this works like the other kinds of rebasing. Changes are computed, and the deltas are applied to a particular branchpoint, one at a time. The difference is that in other rebasing scenarios, the commits being rebased are already in their final form for computing the deltas and live in a separate branch (the one being rebased).

With an interactive rebase, you are generally working in the same branch, making changes to some end portion of the set of commits. Also, you have much more flexibility in how you can modify the commits before Git tries to apply them, rather than just trying to rebase the deltas from their existing versions.

> **WARNING** *Again, the idea here is that the interactive rebase would be used before you pushed the branch targeted for the rebase over to the remote repository. Or, if you had to use the interactive rebase functionality to modify something that was already in the remote repository, you would use the cautions and strategies I've talked about before to manage that kind of scenario.*
>
> *The idea with interactive rebasing is that you are telling Git what to do (leave, modify, delete, and so on) with each commit in a series of commits that exist in the local repository. You can think of it as creating a script or batch file to do modifications on these commits.*
>
> *It is also worth noting that this operation will change SHA1 values for the affected commits. Keep in mind that your history will be different after this, even presumably for commits that weren't explicitly changed themselves.*

Preparation

To begin with, you choose a starting commit. This is the last commit *before* the series that you want to change. Git gives you an opportunity to change all commits *after* this one. As I've discussed in the sections on reset and rebase, you can get the SHA1 value for the commit you want from sources such as the log or the reflog.

Interactive rebasing is done via a file you modify in your default editor. So before you begin, make sure that you have your default editor for Git configured to the application you want to use. Chapter 4 describes how to do this.

Workflow

To begin the interactive rebase operation, you run the git rebase command with the -i option and the starting commit (again, the commit you want to change things after). The command will look like this: *git rebase -i <starting commit>*.

At this point, Git brings up an editor session with a list of the commits that ranges from one after the starting commit to the tip of the branch. Along with these commits, there will be a place to specify an action to be done to each commit. You then modify the action associated with each commit—if you want to do something other than just keep the commit as-is.

Once you have your script or batch file set up to modify the commits the way you want, you save the file and close the editor. (Make sure to close the editor, not just save the file.) Control then returns to Git, which starts processing the sets of actions against the commits in the list.

If any of the actions require input from you (such as rewording a commit message or supplying a new one), Git pauses at that step, and brings up another editor session for the user's input. Sometimes, this can happen so quickly that it appears as if your original editor session is still open. However, if you look at the contents of the file in the editor, you can see the difference.

Once you supply the requested input, the process is the same: save the file and close the editor application. Git then continues on with processing the next commit in the sequence.

Git may also pause the process if it encounters a merge conflict from any of the actions that the interactive rebase causes. If that happens, you can resolve the conflicts in any of the ways that I've described in this chapter, then stage the results, and run the command *git rebase --continue* to continue the operation.

Once all of the processing is done for the series of commits, the operation is done.

Available Actions

Git provides a defined set of actions that you can apply to each commit that is involved in the interactive rebase. Like options to Git commands, these actions can be specified in a long form (spelled out) or a short form (abbreviated by first letter).

The available list of actions includes the following:

- **pick (p)**—Keep this commit as it is.
- **reword (r)**—Keep this commit, but let the user change the commit message associated with it.
- **edit (e)**—When processing this commit, stop and allow for whatever changes the user wants to make. Changes can be made by staging them, and then using the --amend functionality.
- **squash (s)**—Keep the content of this commit, but meld it into the previous commit. When processing this commit, Git stops and prompts, via an editor session, for a new commit message, because multiple commits with multiple messages are now being combined into one.
- **fixup (f)**—Like squash, but don't stop and prompt for a new commit message.
- **exec (x)**—Run the rest of the line as a command in the shell. This action does not take a commit to work against.
- **drop (d)**—Remove this commit from the sequence. This can also be accomplished by just deleting the line with the commit from the file.

Example

Let's look at an example of doing an interactive rebase. Assume you have the results of some previous work with a branch that looks like Figure 9-32.

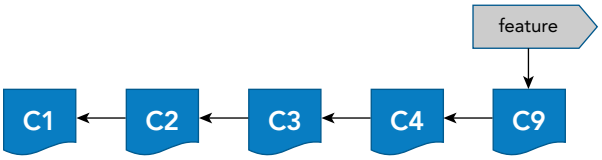


FIGURE 9-32: Beginning state of your branch

A log of this branch would look something like this (the commit names are again used as comments):

```
$ git log --oneline
5b06b1f C9
54d5770 C4
6b77391 C3
27cd1a1 C2
ef77f69 C1
```

You have decided that you want to change some things about the last three commits on this branch: C9, C4, and C3. The easiest way to do that is with an interactive rebase. So, because you always pass the reference to the commit before the first one you want to modify, you start the command telling Git to use the commit three before the current HEAD. (You can also just pass in the SHA1 value of C2.) In this form the command would look like: `git rebase -i HEAD~3`.

This brings up a temporary file in your default editor like the one shown in Figure 9-33.

```
1  pick 6b77391 C3
2  pick 54d5770 C4
3  pick 5b06b1f C9
4
5  # Rebase 27cd1a1..5b06b1f onto 27cd1a1 (3 command(s))
6  #
7  # Commands:
8  # p, pick = use commit
9  # r, reword = use commit, but edit the commit message
10 # e, edit = use commit, but stop for amending
11 # s, squash = use commit, but meld into previous commit
12 # f, fixup = like "squash", but discard this commit's log message
13 # x, exec = run command (the rest of the line) using shell
14 # d, drop = remove commit
15 #
16 # These lines can be re-ordered; they are executed from top to bottom.
17 #
18 # If you remove a line here THAT COMMIT WILL BE LOST.
19 #
20 # However, if you remove everything, the rebase will be aborted.
21 #
22 # Note that empty commits are commented out
23
```

FIGURE 9-33: Temporary file created for scripting the rebase actions

File Format

Notice a couple of things about this file. First, at the top, you have a set of lines with the commits after the one you specified on the command line. Each of these lines has the form,

```
<action> <SHA1 of the commit> <commit message - i.e. comment>
```

For each of these lines (commits), you can modify the default action (pick) to tell Git you want to do something else with this commit. The action is the only thing you modify in these existing lines unless you are deleting a line or adding a separate *exec* action.

Underneath the list of commits are a lot of comment lines (as denoted by the # sign in the first column). These are informational lines. This is Git's way of trying to provide *embedded* help in the temporary files. These lines will not show up in the commit messages or further processing. They are just information for the user and have no bearing on the actual operation.

One thing you may notice about the list of commits is the order they are in: from the oldest to the newest. Around the middle of the comments section, there is a comment that says, *These lines can be re-ordered; they are executed from top to bottom*.

The order in which the commits are listed is consistent with how you would build up a chain in Git, adding newer content each time. However, it is possible to reorder these lines, as the comment says. For this example, I'll just leave the list in the same order.

Choosing Actions

Now, let's tell Git what you actually want to do with these commits as you rebase them. For this example, you'll do a variety of things. First, you'll keep the oldest one, C3, as it is, so you just *pick* that one. Then, you'll delete C4, add an exec step to print out a message, and finally squash C9 into C3. (Note that you always need to have an older commit picked if you're going to squash a newer commit in the sequence.)

Here, I'm using the *x* (exec) action to print out a simple message for illustration purposes. In a real-world situation, you might choose to do something more substantial with an exec, such as some kind of simple testing or validation by running a script.

Figure 9-34 shows your completed script. Git actually stores the resulting script internally as a rebase *TO-DO* file, which also describes it fairly well.

Running the Script

Once you have your edits made to choose the actions you want Git to take during the rebase, you can run the script by saving it and then closing the application. (Note that the application needs to be ended, so that Git will resume and process the saved *to-do* list.)

Git then begins executing the actions from top to bottom. Your first action told Git just to keep C3, so there's no real work there. Then, you told Git to delete C4, shell out and print a message, and squash C9 into C3. As part of the rebase, Git attempts to apply C9 onto the chain. In these cases, it's possible to run into a merge conflict. Here's an example:

```
Executing: echo "Working on squashing C9 into C3"
Working on squashing C9 into C3
error: could not apply 5b06b1f... C9
```

```
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".

Could not apply 5b06b1fa334784daa9e66ac431e9dc4891fc3c89... C9
```

```
1  pick 6b77391 C3
2  d 54d5770 C4
3  x echo "Working on squashing C9 into C3"
4  squash 5b06b1f C9
5
6  # Rebase 27cd1a1..5b06b1f onto 27cd1a1 (3 command(s))
7  #
8  # Commands:
9  # p, pick = use commit
10 # r, reword = use commit, but edit the commit message
11 # e, edit = use commit, but stop for amending
12 # s, squash = use commit, but meld into previous commit
13 # f, fixup = like "squash", but discard this commit's log message
14 # x, exec = run command (the rest of the line) using shell
15 # d, drop = remove commit
16 #
17 # These lines can be re-ordered; they are executed from top to bottom.
18 #
19 # If you remove a line here THAT COMMIT WILL BE LOST.
20 #
21 # However, if you remove everything, the rebase will be aborted.
22 #
23 # Note that empty commits are commented out
24
```

FIGURE 9-34: Edited interactive rebase *to-do* script

Your prompt at this point shows the branch (feature), the state (REBASE-i), and the step you're at in the process (4/4):

```
$ <prompt> (feature|REBASE-i 4/4)
```

You can also run a status at this point to get similar information.

```
$ git status
interactive rebase in progress; onto 27cd1a1
Last commands done (4 commands done):
   x echo "Working on squashing C9 into C3"
   squash 5b06b1f C9
  (see more in file .git/rebase-merge/done)
No commands remaining.
You are currently rebasing branch 'feature' on '27cd1a1'.
  (fix conflicts and then run "git rebase --continue")
  (use "git rebase --skip" to skip this patch)
  (use "git rebase --abort" to check out the original branch)

Unmerged paths:
  (use "git reset HEAD <file>..." to unstage)
  (use "git add <file>..." to mark resolution)
```

```
            both modified:   file1.txt

    no changes added to commit (use "git add" and/or "git commit -a")
```

You can then resolve the conflicts as you would for any other case. Once that is done, you can tell Git to continue the rebase using *git rebase --continue*.

Now that you have resolved the conflict, and told Git to continue, the processing can proceed. Because you used *squash* instead of *fixup* here, Git wants you to tell it what the commit message should be for the *squashed* commit. Prior to starting this process, you had two separate commits with two separate commit messages. So, Git opens up another editor session so that you can tell it the commit message to use for the squashed commit that it's creating (Figure 9-35).

```
1  # This is a combination of 2 commits.
2  # The first commit's message is:
3
4  C3
5
6  # This is the 2nd commit message:
7
8  C9
9
10 # Please enter the commit message for your changes. Lines starting
11 # with '#' will be ignored, and an empty message aborts the commit.
12 #
13 # Date:      Sun Jun 5 12:19:48 2016 -0400
14 #
15 # interactive rebase in progress; onto 27cd1a1
16 # Last commands done (4 commands done):
17 #    x echo "Working on squashing C9 into C3"
18 #    squash 5b06b1f C9
19 # No commands remaining.
20 # You are currently rebasing branch 'feature' on '27cd1a1'.
21 #
22 # Changes to be committed:
23 #   modified:   file1.txt
24 #
25
```

FIGURE 9-35: Screen to enter commit message for squashed commits

Notice a few things about the temporary file that Git opened up here. At the top, Git tells you what the previous commit messages were and asks you what the new commit message for the squashed commit should be. Below that, Git tells you about the interactive rebase that's in progress, how many commands are done, and how many commands are left to do (none in this case).

At this point, you can enter a new commit message for the squashed commits, as shown in Figure 9-36.

You can then save the file with the new commit message, exit the editor, and allow Git to continue. At that point, Git commits the squashed commits and the interactive rebase is completed.

```
[detached HEAD 15e4034] Combining C3+C9 in feature branch
Date: Sun Jun 5 12:19:48 2016 -0400
```

```
 1 file changed, 1 insertion(+), 1 deletion(-)
Successfully rebased and updated refs/heads/feature.
```

If you look at a log of where the branch is after your rebase, you can see the new chain.

```
$ git log --oneline feature
88e6134 Combining C3+C9 in feature branch for 123456
27cd1a1 C2
ef77f69 C1
```

```
1  # This is a combination of 2 commits.
2
3  Combining C3+C9 in feature branch
4
5  # Please enter the commit message for your changes. Lines starting
6  # with '#' will be ignored, and an empty message aborts the commit.
7  #
8  # Date:      Sun Jun 5 12:19:48 2016 -0400
9  #
10 # interactive rebase in progress; onto 27cd1a1
11 # Last commands done (4 commands done):
12 #    x echo "Working on squashing C9 into C3"
13 #    squash 5b06b1f C9
14 # No commands remaining.
15 # You are currently rebasing branch 'feature' on '27cd1a1'.
16 #
17 # Changes to be committed:
18 #   modified:   file1.txt
19 #
20
```

FIGURE 9-36: Adding a new commit message for the squashed commits

However, notice that if you look at a log from the commit that feature pointed to before the interactive rebase, your chain of commits is still there.

```
$ git log --oneline 5b06b1f
5b06b1f C9
54d5770 C4
6b77391 C3
27cd1a1 C2
ef77f69 C1
```

Figure 9-37 shows what your chains of commits look like after the rebase is complete. Note that there is now only one updated commit off of C2 in feature. This is correct because you deleted C4 and squashed C9 and C3 into one. The branch pointer for feature has been moved to your new HEAD after the interactive rebase, but your old chain is still in Git for now. This is what allows you to easily undo or reset back to what you had before the rebase by simply moving the branch pointer.

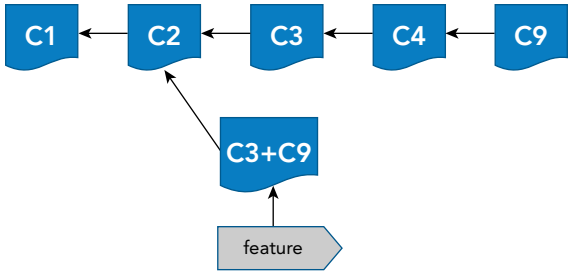


FIGURE 9-37: Your chains of commits after the interactive rebase is completed

Editing Commits

Although I didn't include it in the example, one of the most powerful actions in an interactive rebase is the edit action. This action allows you to go back and update any commit in almost any way you want. When the interactive rebase reaches the edit action and stops, the idea is that you make any changes you want locally (editing or adding files, for example), and then stage those changes into the staging area. You can then use the commit command with the --amend option to amend that commit with the updates from the staging area. Once the changes are committed, you can tell the rebase to continue in the usual way using the --continue option. When stopped for an edit action, the status command shows more detail and suggestions, as shown here.

```
$ git status
interactive rebase in progress; onto a55d7ef
Last commands done (4 commands done):
   x echo "editing C4"
   e 44972d5 C4
  (see more in file .git/rebase-merge/done)
No commands remaining.
You are currently editing a commit while rebasing branch 'master' on 'a55d7ef'.
  (use "git commit --amend" to amend the current commit)
  (use "git rebase --continue" once you are satisfied with your changes)

nothing to commit, working directory clean
```

SUMMARY

In this chapter, I have introduced the concepts and workflow around merging in Git. I discussed the types of merges that Git may do, resolution strategies and options, and how to deal with conflicts and complete or abort merges.

I also introduced the Git rebase and cherry-pick functionality to incorporate history and specific commits into merges. All of your workflow for a merge can also apply to a rebase, and most of it applies to cherry-picks as well.

I covered the workflow used and expected by Git to resolve a merge state for a merge, rebase, or cherry-pick. I also talked about a way to make this process simpler and more user-friendly by using visual merge tools. I presented a brief survey of some visual merge tools that can be used with Git, and discussed how to make them work with the application.

In the Advanced Topics section, I looked at an alternate style for displaying conflicts, and some advanced rebase forms. This included the rebase format that takes three branches, and allows for filtering what's rebased against another branch before making the changes. The interactive rebase functionality allows for creating a script or batch file to modify a series of commits already made to the repository. The rebasing options allow for changing history if needed, but the standard warnings and cautions against changing anything already pushed to the remote side apply here as well.

In the next chapter, you will look at two supporting files that Git uses. The Git Ignore file tells Git which files not to track. And the Git Attributes file tells Git how to process files of specific types. Both of these can play a key role in your local Git environments and workflow.

About Connected Lab 6: Practicing with Merging

Lab 6 takes you through some simple practice with creating merge situations and resolving them. You are encouraged to work through the lab's steps to gain a deeper and hands-on understanding of how merges are handled with Git.

Connected Lab 6

Practicing with Merging

In this lab, you'll work through some simple branch merging.

PREREQUISITES

This lab assumes that you have done Connected Lab 5: Working with Branches. You should start out in the same directory as that lab.

STEPS

1. Starting in the same directory that you used for Connected Lab 5, make sure you don't have any outstanding or modified files (nothing to commit). You can do this by running the status command and verifying that it reports "working directory clean."

   ```
   $ git status
   ```

2. If not already on the master branch, switch to it with *git checkout master*. Create a new one-line file.

   ```
   $ echo "Initial content" > file5.c
   ```

3. Stage and commit the file on the master branch.

   ```
   $ git add .
   $ git commit -m "adding new file on master"
   ```

4. Start up gitk if it's not already running.

   ```
   $ gitk &
   ```

5. Create a new branch, but don't switch to it yet. (You can use whatever branch name you want.)

   ```
   $ git branch newbranch
   ```

6. Change the same line in the new file (still on the master branch).

```
$ echo "Update on master" > file5.c
```

7. Stage and commit that change (still on the master branch).

```
$ git add .
$ git commit -m "update on master"
```

8. Switch to your new branch.

```
$ git checkout newbranch
```

9. On the new branch, make a change to the same line of the same file.

```
$ echo "Update on newbranch" > file5.c
```

10. Stage and commit the file with the change on the new branch.

```
$ git commit -am "update on newbranch"
```

11. Switch back to the master branch.

```
$ git checkout master
```

12. Merge your new branch back into the master branch. (Git attempts to merge the new branch into the master branch.) You will end up with a merge conflict after this.

```
$ git merge newbranch
```

13. Check the status of your files in Git. Note the information that Git provides to you about the conflict.

```
$ git status
```

14. Look at the local file and note the conflict markers.

```
$ cat file5.c
```

15. Resolve the conflict in the file in the working directory. (For simplicity, you can just write over it to simulate that the conflict has been resolved.)

```
$ echo "merged version" > file5.c
```

16. Stage and commit the fixed file. Note that this has to be done as two separate steps since this was the resolution to a merge conflict.

```
$ git add .
$ git commit -m "Fixed conflicts"
```

17. Check the status to make sure the merge issue is resolved.

```
$ git status
```

18. Refresh/reload gitk and look at the changes.

19. You're done with your new branch, so delete the branch.

```
$ git branch -d newbranch
```

20. Refresh/reload gitk and look at the window showing the branches to see how the Git repository looks different.

10

Supporting Files in Git

WHAT'S IN THIS CHAPTER?

- Purpose of the Git attributes file
- Scope and use cases for the Git attributes file
- Format of the Git attributes file
- Purpose of the Git ignore file
- Scope and use cases for the Git ignore file
- Format of the Git ignore file

In this chapter, I explore two types of supporting files that allow users to customize how Git interacts with different kinds of content. The *Git attributes* file allows you to define settings to apply to certain operations for particular files or file types. The *Git ignore* file allows you to tell Git which files it should ignore and not try to manage. I cover the intent, usage, scope, and format for both types of files.

Having these two files with each repository is a best practice for Git. There are also many customized versions that have been created by users for different types of work. For example, an attributes file for Java development could include specifications on line endings for *.java files. The corresponding ignore file could include lines to tell Git not to track or manage *.jar, *.war, and other files, because those are generated files in the java workflow.

Over the years, Git users doing different kinds of development have created versions of these files that have worked well for them. In the spirit of open source, users commonly contribute the files that they have used back to public places on the Internet. One popular site for these contributions from users is GitHub. Normally, GitHub is thought of as a place to host Git repositories. However, it is also a place for users to share supporting items for development.

You can find some sample Git attributes files for different languages at `https://github.com/Danimoth/gitattributes` and sample Git ignore files for different kinds of development at `https://github.com/github/gitignore`.

Now, let's dive into the specifics of an attributes file.

THE GIT ATTRIBUTES FILE

One thing that is not clear with Git out of the box is a way to specify how to handle specific file types. The most obvious example of this is being able to tell Git which files are binary. (Note that Git has an algorithm built-in to try and detect if a file is binary, but it cannot be claimed to be 100% accurate.)

To provide this kind of functionality, Git supports specifying options about how to treat different kinds of files using a *Git attributes* file. As the name implies, this file allows users to specify attributes for files that Git manages. These attributes serve as specifications that drive or modify the behavior of particular Git operations when those operations are done on the matching files. For example, if a file or file type is noted as binary, that tells Git not to do the usual diffing and EOL processing against it.

What Can You Do with This File?

A Git attributes file can be used for a number of purposes. Here are a few of the main applications:

- Specifying which files or file types Git should treat as binary
- Specifying how to handle line endings for particular files or file types
- Specifying unique filters to associate with files or file types to perform custom operations

I'll spend some time talking about these uses in the following pages. First, though, it's useful to understand the different places that versions of this file can exist—and how their location determines which files get which attributes.

The Scope of Git Attributes

In your working directory, at the granularity of a subdirectory, you can create a Git attributes file. These files are named *.gitattributes*. Having these exist in the working directory is useful because this file can be committed into your local repository, then pushed to the remote repository, and reside with your code. So, any user or process that clones this repository will get the same .gitattributes file telling Git how to handle files in this repository.

The other useful part about having a Git Attributes file in your working directory is that, for things like line endings, the Git attributes file overrides individual configuration values. This means you can specify how to handle line endings in the Git attributes file, regardless of how a user may have their configuration values set. Because the Git attributes file is pulled in when the project is cloned, the project will always use the settings from the file. Thus, you have consistent line-ending behavior across all users of the repository.

> **TIP** *Due to the consistency that you gain from using a Git attributes file, this is a best practice for every project in Git that is intended for multiple users.*

Similar to how you have local, global, and system configuration scopes, Git attributes files can also have different scopes. The same file format as a .gitattributes file can be put in the .git directory—in .git/info/attributes (where *attributes* is a filename, not a directory). This will have higher precedence than a .gitattributes file in the working directory.

Within the subdirectories of the working directory, there can be individual .gitattributes files (one per directory). When it comes time to evaluate them, Git will use the attributes in the .git/info/attributes file first (if present) and then the .gitattributes file it finds that's closest to the path being evaluated.

So, suppose I have a structure like dir1/dir2/dir3 and I have .gitattributes files in all three subdirectories. If I am working with a file in dir3, Git will first use any attribute information from .git/info/attributes, then any information from the .gitattributes file in dir3, then from the one in dir2, and finally from the one in dir1.

Beyond that, you can have a global Git attributes file. This one, like global configuration values, applies to all repositories for a user. The location of this file can be set explicitly by setting the core .attributesFile configuration value. This has a default value of $XDG_CONFIG_HOME/git/attributes. (It is rare to have the $XDG_CONFIG_HOME reference set. See the note in this chapter on "Git's Fourth Configuration File.") If the $XDG_CONFIG_HOME reference is not set, then you fall back to the more common $HOME/.config/git/attributes file (where $HOME is the user's home directory).

GIT'S FOURTH CONFIGURATION FILE

Technically, there is a fourth configuration file type that Git supports.

If you look at the man page for git config, you'll see it lists another possible configuration file as $XDG_CONFIG_HOME/git/config. That page defines it as follows:

"Second user-specific configuration file. If $XDG_CONFIG_HOME is not set or empty, $HOME/.config/git/config will be used."

XDG stands for *X Development Group*, the previous name of FreeDesktop.org. FreeDesktop.org is intended for code and discussion of software projects focused on interoperability and sharing of technology for X Window System desktops such as GNOME and KDE.

If this file exists, it fits in between the system configuration file and the global configuration file. Using this location allows for having a configuration that conforms to the XDG recommendations on systems where those recommendations are used.

However, unless you have a specific need for this file on a particular system, it is safer and simpler to not use it.

That just leaves the system-level Git attributes file—for all users on a system. Like system configuration files, this file resides in the etc area, such as /etc/gitattributes.

Table 10-1 summarizes the scopes for Git attributes files, starting from the highest priority to the lowest.

TABLE 10-1: The File Scope for Git Attributes

SCOPE	NAME AND LOCATION	STORED WITH PROJECT	USE/NOTES
Internal	.git/info/attributes	No	Overrides all others but not stored with project (user-specific)
Local	subdir/.gitattributes	Yes (committed)	Defines attribute handling for files in this <subdir> and any below it that don't have their own .gitattributes. This file is committed into source control and resides with the project. It overrides certain configuration values.
Global	Usually $HOME/.config/git/attributes or as specified by core.attributesFile configuration value	No	All repositories for a user
System	etc/gitattributes	No	System attributes for all repositories if not overridden by one of the others

The File Format

The file format for a Git attributes file is fairly simple. There are only a few rules:

- Lines are composed of filenames or patterns followed by attribute specifications.
- Any line starting with a # character is a comment.
- Lines that specify attribute settings have a format of

  ```
  <file pattern or filename> <attribute and setting>
  ```

Some examples of a file pattern or filename include *.obj, foo.txt, and a*.java.

On each line of a Git attributes file, attributes can be defined in multiple ways. Table 10-2 summarizes them.

TABLE 10-2: Options for Specifying Attributes

SETTING	FORMAT	EXAMPLE	MEANING
Set	Attribute name by itself	*.obj binary	Attribute is turned on (all object files are binary in the example).
Unset	Attribute name preceded by minus sign	*.java -crlf	Attribute is turned off (no java files should have CRLFs in the example).
Configured (set to a value)	Attribute name = value	eol = crlf	Attribute is configured with a specific value (set line endings to crlf on checkout for matching files in the example).
Unspecified	Attribute isn't specified for any paths		Behavior falls back to what Git would do without the attributes file.

Here are a couple of other points to remember about the file format:

- Later lines that match override earlier lines.
- More than one attribute can be specified (exist on the same line) for a file or file type.

Common Use Cases

In this section, I'll survey some common use cases for a Git attributes file. First, though, there are some things you need to consider when creating or amending one of these files.

Putting together a Git attributes file is a combination of several steps:

1. Deciding on the scope: Should this be managed with the repository (reside in the working directory); be global (reside in $HOME); apply to the working directory, but not be managed (reside in git/attributes); or reside in one of the subdirectories?
2. Identifying the file or pattern that you want to apply an attribute to.
3. Choosing the appropriate attribute.
4. Choosing whether the attribute should be *set*, *unset*, *set to a value* (string), or *unspecified* to accomplish what you want.

5. If the attribute is set to a custom value or a filter is specified, defining the value or filter for Git—this includes creating external programs if needed and setting configuration values.
6. If custom supporting pieces are needed (as in step 5) and this setup is intended to be pushed into a remote repository for use by others, ensuring that the supporting pieces are accessible and any additional needed configuration is well documented.

Now, let's look at a couple of common use cases and useful tricks to illustrate what you can do with the file. You will implement these use cases through the use of attributes. Although not explicitly stated each time, the idea is that these attributes apply to the file or patterns on the corresponding line in the Git attributes file.

Use Case 1: Identifying Files as Binary

This one is fairly simple. To tell Git that a file is binary, simply set the binary attribute in the Git attributes file. For example, **.exe binary* . *Binary* here actually corresponds to the deprecated settings of *-crlf* and *-diff*. So, if you think about the dash (-) on the front of those attributes as meaning *don't do it*, you could interpret binary as *don't try to convert or fix eol issues* (-crlf) and *don't try to execute or print a diff* (-diff) for executable files.

Use Case 2: Specifying Handling for Line Endings

In this case, you have several attributes to choose from:

- **text**—This attribute has to do with line normalization and indicating text file types. Here are some examples of how it can be specified, along with their meanings.
  - *text* (setting the value) tells Git to normalize line endings and that files like this are text files.
  - *-text* (unsetting the value) tells Git not to attempt to normalize line endings.
  - *text=auto* (set to value) tells Git to normalize line endings if it thinks the file is text. (To determine this, Git is checking for a NULL value in a significant part of the file. If the NULL value is found, the file is considered to be binary.)

  If text is not specified, then Git falls back to using the core.autocrlf value as configured by the user.
- **eol**—This attribute tells Git which style of line normalization to apply. Its presence tells Git that this is a text file.
  - *eol=crlf* (set to value)—llike *autocrlf=true*, this value tells Git to normalize files (to just LF) at add or commit, and insert CRLFs on checkout.
  - *eol=lf* (set to value)—like *autocrlf=input*, this value tells Git to normalize files (to just LF) at add or commit, and leave as LFs on checkout.

NOTE *If you enable text=auto to normalize to LFs in an existing repository, it is a good idea to verify whether any files in that repository need to be normalized from their current state (for example, if they have CRLF). The help for .gitattributes has a sequence of commands that describe how to do this.*

NOTE *As long as I'm on the subject of line endings and potential normalizing, this is a good place to explain another configuration value that has to do with line endings: core.safecrlf.*

The purpose of core.safecrlf is to prevent, or alert users to, irreversible line-ending conversions when settings are used that enable conversions like core.autocrlf.

To understand irreversible*, think of it this way, as paraphrased from the Git help text: When you commit a file and it goes through line-ending conversion, if you then check it out, you should get back the exact same file you had before you committed it.*

If the core.autocrlf setting will result in you getting back a different file right after you check it in, then if core.safecrlf is set to true*, Git will not allow the change and will report a fatal error. If core.safecrlf is set to* warn*, then Git will just issue a warning.*

Realistically, the potential irreversible situation should only arise if a text file has mixed line endings (a bad thing)—because Git can't reproduce mixed line endings—or if Git has incorrectly detected that a binary file is text (highly unlikely) and tries to modify characters in it to convert line endings.

Use Case 3: Creating a Custom Filter

If you simply define the attribute as *filter=<name>*, Git assumes you are defining a custom filter called *<name>*. This allows you to have two filtering actions, known as *smudge* and *clean*.

Smudge is essentially a way to say, "For the matching items in the Git attributes file, run a set of code (a filter) when those items are checked out of Git." Clean is the same, but it runs designated code when the matching items are committed back into Git. Figure 10-1 shows where these actions fit into your Git model.

Git assumes that the name you provide to the filter actually refers to a filter driver. A *filter driver* is a program or set of commands that are defined to execute for the smudge and clean operations when this filter is run. To map the filter names to the actual code that is executed, you just use the standard git config operation.

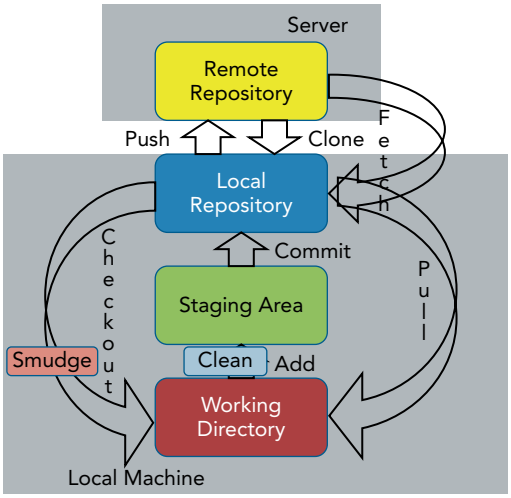


FIGURE 10-1: The Git model with smudge and clean filters

Let's take a look at a simple example to illustrate how all of this works. Suppose you have a couple of HTML files that you use as a header and footer across multiple divisions in your company. They contain a placeholder in the form of the text string %div to indicate where the proper division name should be inserted.

You want to use the smudge and clean filters to automatically replace the placeholder with your division name (ABC) when you check the file out of Git, and make sure to set it back to the generic version if you make any other changes and commit those changes.

The following listing shows my two source files:

```
$ cat div_test_header.html
<H1>Running tests for division:%div</H1>

$ cat div_test_footer.html
<H1>Division:%div testing summary</H1>
```

To make the appropriate transformations, I'm going to create a custom filter that will have associated *clean* and *smudge* actions. I'll call this custom filter insertDivisionABC.

This filter will simply use the sed command to substitute ABC for %div, or vice-versa, as required. To define this filter and its actions for Git, you can use the config command as follows:

```
$ git config filter.insertDivisionABC.smudge "sed 's/%div/ABC/'"
$ git config filter.insertDivisionABC.clean "sed 's/ision:ABC/ision:%div/'"
```

These commands say, "When you check out these files, replace the occurrence of %div with ABC. And, if you commit a changed version of this file, set it back to the generic form (with the %div) where you had ABC." (I include a part of *division* here in the replacement string just to be more precise.)

With this form of the config command, the parts after git config are organized as [item class, item name, subitem, subitem value]. So, if you were to look at the actual config file after these commands, you would find the following section:

```
$ cat .git/config
        ...
 [filter "insertDivisionABC"]
        smudge = sed 's/%div/ABC/'
        clean = sed 's/ision:ABC/ision:%div/'
```

The last part of connecting everything together is to create a (or append to an existing) Git attributes file with a line that tells Git to run your filter for files matching the desired pattern. The simplest kind of file would be one such as created by the following command:

```
$ echo "div*.html filter=insertDivisionABC" > .gitattributes
```

The presence of a .gitattributes file with the line `div*.html filter=insertDivisionABC` tells Git, "For files matching the pattern, run the specified filter." This then points to the commands you have configured, depending on whether a checkout or add is being done.

After a checkout with the attributes file and the filter is in place, your local file contents would look like the following output:

```
$ cat div_test_header.html
<H1>Running tests for division:ABC</H1>

$ cat div_test_footer.html
<H1>Division:ABC testing summary</H1>
```

Suppose that you now make changes to the local files, adding the text *full* in one and *all* in the other. A diff will take into account the filter and just show you the differences without the substitution being applied.

```
$ git diff
diff --git a/div_test_footer.html b/div_test_footer.html
index 8a5bb93..60e32b7 100644
--- a/div_test_footer.html
+++ b/div_test_footer.html
@@ -1 +1 @@
-<H1>Division:%div testing summary</H1>
+<H1>Division:%div full testing summary</H1>
diff --git a/div_test_header.html b/div_test_header.html
index 124625e..fd1aeb6 100644
--- a/div_test_header.html
+++ b/div_test_header.html
@@ -1 +1 @@
-<H1>Running tests for division:%div</H1>
+<H1>Running all tests for division:%div</H1>
```

If you now add the files, a status command shows them as staged and modified.

```
$ git add div*.html

$ git status
```

```
On branch feature
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   div_test_footer.html
        modified:   div_test_header.html
```

A diff command shows no differences. If you cat the local versions of the files, they still show the substitution from the smudge.

```
$ git diff


$ cat div_test_footer.html
<H1>Division:ABC full testing summary</H1>

$ cat div_test_header.html
<H1>Running all tests for division:ABC</H1>
```

You know that the clean filter should have removed your substitution, but the question is, "How do you verify that?" There is a trick you can use with the show command to see the version in the staging area—*show :0:<filename>*. Using this command, you can verify that the clean filter was applied when the files were staged.

```
$ git show :0:div_test_footer.html
<H1>Division:%div full testing summary</H1>

$ git show :0:div_test_header.html
<H1>Running all tests for division:%div</H1>
```

Use Case 4: Tricking Git Merge into Ignoring Files or Paths

For this use case, you can use a trick to tell Git not to merge a file or path when a merge operation is run on the larger set of files. This is not necessarily a common, intended use case, but it can be very useful and illustrates how to use filters on attributes to accomplish custom tasks.

Another type of attribute available in the Git attributes file is *merge*. Normally, the merge attribute, if specified, is resolved to text or binary. However, using a filter in another way, you can specify a program or command to run when the merge attribute is specified. Normally, handling the merge filter would be similar to defining a custom mergetool (see Chapter 9) that expects certain arguments and performs certain behaviors.

For your purposes here, though, you just need to specify a tool or command that returns a value of true. This is because Git expects any kind of merge filter like this to return true to indicate that the custom merge behavior was successful and you are done. Another characteristic of the merge filter implementation is that when the merge has been completed, the result is stored in the original filename. So, if you just return true here, this is effectively a NO-OP as far as changing the file is concerned. However, Git thinks it has done its job and processed this file as part of the merge.

To set this functionality up, in the Git attributes file, you can define a custom filter for the merge attribute to use. One common name that is often used for this purpose is *ours* to correspond with the merge option for the recursive strategy and the Ours strategy. For your purposes here, to be a little clearer, you'll use the name *unchanged*.

Then, to make this all work, you set a configuration value that tells Git to define a filter driver for the merge attribute value of unchanged. This filter driver just has to return true (assuming you have a *true* command available on your system). The command to configure this value would look like this:

```
$ git config [--global] merge.unchanged.driver true
```

Then, in your Git attributes file, you can just specify *merge=unchanged* for the file or path.

```
$ cat .gitattributes
filename.ext merge=unchanged
```

When Git is running a merge that includes this file or path, it assumes from the attributes file that it should run the filter program named unchanged, which then just evaluates to true. Git is done at that point, and so the file or path is not changed.

Note that this trick depends on a particular configuration that needs to be shared and set if other users will use this Git attributes file in this way.

Other Attributes

There are other attributes I haven't covered that are available to use in a Git attributes file. Two attributes that you may encounter are ident and diff:

- **ident**—This attribute causes a substitution to occur if the string *Id* is found in a file on checkout. In other source management systems, this kind of substitution results in items like userids, dates, and so on being substituted. In Git, this is less useful because the string is just expanded to include the SHA1 value as *$Id: <sha1 value> $*. On checkin, the substitution is reverted.
- **diff**—This attribute describes how diffs are generated for the matching file or file types. In short, if it is set (*diff*), the diff is generated as text. If it is unset (*-diff*), it is treated as a binary file when a diff is attempted. If it is unspecified, then the default diff behavior is performed. And if it is set to a string, that string is treated as a filter driver that can be defined to do whatever operations are desired when a diff for the matching files or file type is invoked.

Example File

Combining all of these attributes into an example Git attributes file might look like this:

```
$ cat .gitattributes
# Default is to detect text files and perform normalization to LF
* text=auto

# Treat exe files as binary
*.exe binary

# Don't insert CRLFs in our sh files
*.sh -crlf

# Set eol sequence to CRLF for *.txt files
*.txt eol=crlf

# Don't merge our home page file
```

```
index.html merge=unchanged

# Use our custom smudge and clean filters for these files
div*.html filter=insertDivisionABC

# Update text files in the misc directory
misc/*.txt filter=change_text
```

Note that you have the text=auto attribute setting for everything as your first entry. This acts as a sensible default for normalizing files that Git detects as text. This setting can be overwritten for specific files and file types by later entries in the file.

The last entry here is one I didn't use in the text. However, it illustrates the idea that subdirectory paths can be used to restrict attributes. In this case, the *change_text* filter will be applied to files with the *.txt* extension, but only files in the *misc* subdirectory. Files with the same extension in other subdirectories will not have the filter applied.

Getting Attribute Information for Files

When your Git attribute files start to become large or complicated, or you are dealing with many of these files, you may want to have a way to quickly summarize which attributes are being applied to which files without having to read through the files. Fortunately, Git provides a plumbing command designed to do just that: *check-attr*. (Recall that plumbing commands are hyphenated, action-object names.)

The format for the check-attr command is pretty simple. There are basically two forms.

```
git check-attr [-a | --all | attr...] [--] pathname...
git check-attr --stdin [-z] [-a | --all | attr...]
```

For each pathname, the command displays information about what attributes apply to it, and whether they are set, unset, unspecified, or set to a specific value. The attributes can be filtered by a particular attribute, or *--all* to see all of them. Some simple examples from your custom filter setup are shown here.

```
$ git check-attr filter div*.html
div_test_footer.html: filter: insertDivisionABC
div_test_header.html: filter: insertDivisionABC

$ git check-attr div*.html --all
div_test_footer.html: filter: insertDivisionABC
div_test_header.html: filter: insertDivisionABC
```

Now you'll look at the other main support file that Git repositories use, Git ignore. This file tells Git what not to track.

THE GIT IGNORE FILE

In addition to being able to specify particular attributes for files or file types, Git also provides a way to ignore particular files and file types. Items to ignore can be specified in a text file called a *Git ignore* file.

Ignore in this case means to exclude from tracking and processing. In other words, Git does not attempt to manage or change the indicated files or directories. They are automatically ignored if they exist in an ignore file.

The Scope of Git Ignore

At the directory granularity, you can create a Git ignore file as a file named *.gitignore* in your working directory. This is useful because this file can be committed into your local repository and then pushed to the remote repository. It resides in your repository, with your code. So, any user or process that clones this repository will get the .gitignore file that tells Git what to ignore.

Similar to how you have local, global, and system configuration scopes, Git ignore files can also have different scopes. The same file format can be stored internally in the .git directory, in .git/info/exclude (where exclude is a filename, not a directory). This is not staged and committed with the source files because it is in the internal repository, so it's useful for personal ignore preferences that wouldn't apply to other users. It also has a lower precedence than a .gitignore file in the working directory.

Within the subdirectories of the working directory, there can be individual .gitignore files (one per directory). When it comes time to evaluate them, Git uses the Git ignore file in the current directory first and then the Git ignore file it finds that's closest to the path being evaluated.

So, suppose I have a structure like dir1/dir2/dir3 and I have .gitignore files in all three subdirectories. If I am working with a file in dir3, Git first uses any ignore information from there, then from the file in dir2, and finally from the file in dir1.

Beyond that, you can have the *internal* Git ignore file in .git/info/exclude, and also a global Git ignore file. That one, like global configuration values, applies to all repositories for a user. The location of this file can be set explicitly by specifying the core.excludesFile configuration value. This has a default value of $XDG_CONFIG_HOME/git/ignore. (It is rare to have the $XDG_CONFIG_HOME reference set. See the note in this chapter on "Git's Fourth Configuration File.") If the $XDG_CONFIG_HOME reference is not set, then you fall back to $HOME/.config/git/ignore.

Table 10-3 summarizes the scopes for Git ignore files, starting from the highest priority to the lowest.

TABLE 10-3: Scopes and Precedence for Git Ignore Files

SCOPE	NAME AND LOCATION	STORED WITH PROJECT	USE
Local	subdir/.gitignore	Yes (committed)	Defines ignore values for files in this subdirectory and any subdirectories below it that don't have their own .gitignore file. This file resides with the project.

continues

TABLE 10-3 *(continued)*

SCOPE	NAME AND LOCATION	STORED WITH PROJECT	USE
Internal	.git/info/exclude	No	For settings that should apply to the repository, but not be stored with it (that is, files resulting from custom workflow, experimentation, and so on)
Global	Usually $HOME/.config/git/ignore or as specified by core.excludesFile configuration value	No	Files to ignore across all repositories for a user (for example, the user's text editor's backup files, which might be consistent across all repositories for one user but different for another user)

The File Format

The file format for a Git ignore file is fairly simple. Here are the basic features:

- Blank lines can be used as separators for readability.
- Any line starting with a # character is a comment. (Add a backslash in front of the # if there is a filename that actually starts with #.)
- If a line ends with a forward slash (/), Git recognizes that forward slash as a directory. It matches the directory path and all things under the directory, but not any files or links with the same name as the directory.
- Without a forward slash (/), Git tries to match up any paths and patterns specified in the file relative to the repository path.
- Two consecutive asterisks (**) generally mean *match 0 or more subdirectory levels*. For example, b/**/e matches b/e, b/c/e, b/c/d/e, and so on.
- An exclamation point (!) at the start of the line tells Git to negate the pattern. Any file in that pattern that was previously excluded (ignored) becomes included again—unless a parent directory has been excluded, which prevents the file's re-inclusion.

The use cases for this last kind of format deserve some more explanation.

Use Cases for Pattern Negation

At first glance, the idea of pattern negation might seem strange in a Git ignore file. If you are going to include the file, why list it and negate it instead of just not putting it in the ignore file in the first place?

There are two typical uses for this format. One is to override a more general exclusion in a specific path, and the other is to allow targeted inclusion in cases where you want to exclude most other items.

Overrides

Suppose that you have a long *global* gitignore file that includes a rule to ignore backup files that your editor produces.

```
$ git config --list --global | grep excludes
core.excludesfile=/Users/dev/.config/git/ignore

$ cat /Users/dev/.config/git/ignore
# Ignore bak files no matter where in the tree
*.bak
```

Now, in one of your repositories, you decide that the information is critical and it would be useful to track and manage the backup files in this one area. Rather than re-creating a local copy of the entire global Git ignore file, you can just create the internal ignore file and have it override that one line.

```
$ cat /Users/dev/myrepo/.git/info/exclude
# Track *.bak files for this repository
!*.bak
```

EXCLUDES VERSUS EXCLUDE

Note that the configuration setting for the global ignore file is core.excludesFile (where exclude ends with an *s*), while the internal ignore file is .git/info/exclude (without an *s*).

Now, instances of the backup files in this repository are tracked on your system. Note that since neither the global ignore file or the one internal to the repository is committed into Git, this will not affect other users that clone the repository.

Targeted Inclusion

The use case here would revolve around wanting to exclude most items in an area, but include a small set of targeted items. One approach is to try and list everything that needs to be ignored down to a low level of detail, leaving only the small subset of items to be included out of the file.

However, a simpler approach is to list the higher-level areas in the ignore file and then create a negated pattern for each level down to the set of items that you want included. Effectively, you are cutting a hole through the hierarchy by negating the explicit subpaths to the desired set of items. I said earlier that it's not possible to negate a file if the directory above has been excluded. This trick of negating the directory paths along the way allows you to get around that limitation.

As an example, suppose you have the following structure in your local Git repository:

```
myrepo
|   .gitignore
|
└──subdir1
    |   filea.txt
    |   fileb.txt
```

```
        │
        └──subdir2
                file1.txt
                file2.txt
```

And you have the following text in your .gitignore file:

```
$ cat .gitignore
# ignore all of the things under this directory
subdir1/*
# except for this child directory
!subdir1/subdir2
# but ignore everything under it
subdir1/subdir2/*
# except for this file
!subdir1/subdir2/file1.txt
```

If you perform an add operation, because you have created a path for the one file, you'll have it and the .gitignore file staged. (The .gitignore file will be staged because it's in the current directory.)

```
$ git add

$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   .gitignore
        new file:   subdir1/subdir2/file1.txt
```

Getting Ignore Information for Files

Like the *check-attr* plumbing command to get attribute information for paths, Git also provides a plumbing command to get ignore information and status for paths. The command is named *check-ignore*.

The format for the check-ignore command is pretty simple; there are basically two forms.

```
git check-ignore [options] pathname
git check-ignore [options] --stdin
```

For each pathname, the command displays information about whether that file matches a specification in the set of available Git ignore files that affect the repository. This command reports back on matches for both valid ignore rules and negated ignore rules for unstaged files—printing the filename if there is a match.

If you want more detailed information, you can add the --verbose option. This creates output in the following format:

```
<source> <COLON> <linenum> <COLON> <pattern> <HT> <pathname>
```

where <source> refers to the location of the Git ignore file that matched, along with the particular line number as noted in <linenum> and the <pattern> that matched. This is followed by the pathname that was provided for the command that matched.

```
git check-ignore --verbose *.bak
/Users/dev/.config/git/ignore:2:*.bak   *.bak
```

SUMMARY

In this chapter, I've covered the ideas, scope, format, and practical uses for two supporting files that you can use to tell Git how to deal with specific content. These files can exist in different parts of a Git environment—primarily stored with the files in the repository, or stored internally in the .git repository space. Instances of these two supporting files that are stored in the repository are part of the set of files that everyone gets when they clone. This then provides a consistent specification regardless of each user's environment and personal configuration because they will get it automatically as part of the clone. Instances of these two supporting files stored only within the .git directory apply only to the current user.

A Git attributes file allows you to specify different attributes that dictate how certain operations behave for files that match specified patterns or names. It is primarily used for specifying which items are binary, specifying line endings, and creating custom filters. The custom filters can be one or both of two forms: a smudge operation is a filter that takes place when a file is checked out of a repository; a clean operation is a filter that takes place when a file is checked in to Git.

A Git ignore file specifies files that Git should not track or manage. Examples might include generated files or backup files. Ignore files that are stored with the source files in the repository provide a consistent set of items to ignore for anyone who clones the repository. An exclude pattern can be negated to override a previous ignore or filter a larger set to include a more precise subset.

Example attribute and ignore files for many different types of development have been created and contributed back to public sites like GitHub. They are available to download from those sites and use as starting points.

In the next chapter, you'll look at some other useful commands for manipulating files in Git that are similar to operating system commands. You'll also explore some advanced commands that are useful for performing functions like finding out where a problem was first introduced and teaching Git how to automatically resolve complex merge scenarios.

11

Doing More with Git

WHAT'S IN THIS CHAPTER?

- Stashing work in progress
- Learning about Git commands to remove and rename files
- Searching through content and logs in Git
- Creating patches and external archives of Git content
- Cleaning up working directories and repositories
- Adding notes to commits already in the repository
- Using filter-branch to modify repositories
- Using bisect to find where something was first introduced
- Using rerere to remember and replay resolutions to merge conflicts

So far, most of this book has centered around the Git workflow and the more common source management operations you need to be able to work with it. In this chapter, I cover some less common—but nonetheless, very useful—commands that are available in Git. These are commands that you may not use very often, but when you need them for a specific purpose, it's good to understand how to use them. I begin with some commands for working with file and directory organization in your local environment.

MODIFYING THE LAYOUT OF FILES AND DIRECTORIES IN YOUR LOCAL ENVIRONMENT

When working with your OS, there are basic applications and commands for simple workflows, such as creating and locating files, seeing what's in the file system, and switching

between directories. However, there are also supporting commands for doing things like copying, renaming, and deleting content.

Git also provides these commands for content that it manages. In this section, you'll look at some of these layout commands in Git, with a focus on three commands in particular: *stash*, *mv*, and *rm*. These commands also introduce new interaction with the staging area. I discuss how this interaction works and what it means for the user.

stash

Imagine you are a developer working on a new set of code for a feature or bug fix in a very specific environment that you have set up just for this purpose. You are in the middle of making changes when your boss tells you to drop everything and start working on a higher-priority change that requires the same environment.

To manage this change in direction, you might create a backup directory somewhere and copy your changes that are in progress to that backup area, with the idea of retrieving them later. You could then set your environment back to the way it was before you started working on those changes, so that you could focus on the new task.

Chances are, you've probably encountered some variant of this scenario in your working life, even if you aren't a developer. When working in a Git environment, the Git stash command is your best option in a situation like this. The syntax for the Git stash command is as follows:

```
git stash list [<options>]
git stash show [<stash>]
git stash drop [-q|--quiet] [<stash>]
git stash ( pop | apply ) [--index] [-q|--quiet] [<stash>]
git stash branch <branchname> [<stash>]
git stash [save [-p|--patch] [-k|--[no-]keep-index] [-q|--quiet]
           [-u|--include-untracked] [-a|--all] [<message>]]
git stash clear
git stash create [<message>]
git stash store [-m|--message <message>] [-q|--quiet] <commit>
```

Saving Uncommitted Changes

The stash command saves a copy of your uncommitted changes in a queue, off to the side of your project. By *uncommitted changes*, I mean items in either the staging area or the working directory that have been modified but not committed to the local repository. This state could be due to a couple of reasons, but most commonly it would be because either

- files have been modified since the last commit; or
- the last commit specified a subset of files or directories to commit rather than all eligible ones.

In either case, the repository does not have the current version of those remaining files. The most current version is in the working directory or staging area. As such, they are *uncommitted*.

Each time the stash command is invoked and there is uncommitted content (since the last stash command), Git creates a new element on the queue to save that content. That content can be in the staging area, in the working directory, or both.

What's Left in Your Local Environment?

One of the questions that comes up quickly when using the stash command is what state the local environment is in after running a stash. The short answer is that it's set back to the state it was in immediately after the last commit (pointing to HEAD). After creating the stash and saving the uncommitted content, Git is basically doing a git reset --hard HEAD operation. However, because you have the stash, you haven't lost your uncommitted changes.

Stashing Changes

The default syntax for creating a stash in Git is *git stash <options>*.

Figures 11-1 and 11-2 show a visual representation of doing a default stash (without any options).

In Figure 11-1, you have the local environment model that I am using throughout the book. In this case, the asterisk (*) denotes a change that has been made in the working directory and staged in the staging area since the last time you did a commit.

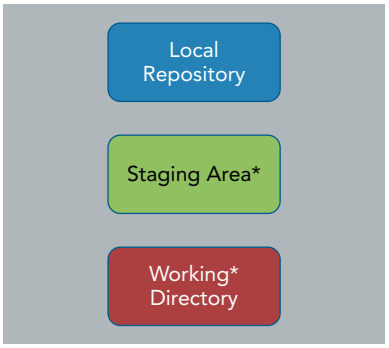


FIGURE 11-1: Local environment with an uncommitted change

Now, you run your stash command:

```
$ git stash
```

Figure 11-2 shows what happens conceptually. Git creates a stash of the staging area and working directory with the uncommitted change. It also resets the local environment back to the last commit (HEAD).

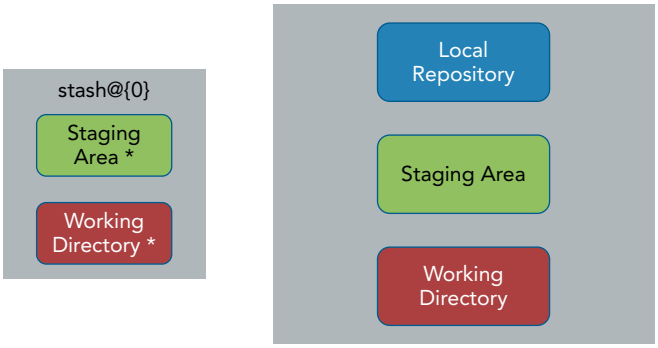


FIGURE 11-2: After the initial stash

Now you have your in-progress changes stored in the stash, and your local environment is clean (in sync with the last commit). This is the basic operation. Now, let's look at a couple of options for stash that can be helpful.

Including Untracked Files

Recall from the discussion on git status in Chapter 6 that when talking about a file's relationship with Git, you can broadly say it's either *tracked* (added to Git and being managed by it) or *untracked* (not yet added to Git).

When stashing content with Git, by default, it ignores untracked files. In order for Git to stash untracked files, it is necessary to include the *-u* (*--include-untracked*) option. This option includes untracked files from the working directory in the stash.

Building on the current example, suppose that you've made another set of changes and staged them, as indicated by the double asterisks (**) in Figure 11-3. Along with that change, you have a new file that you've created, as indicated by the number sign (#).

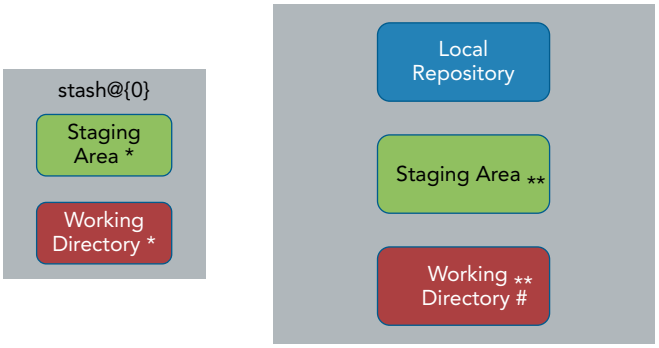


FIGURE 11-3: Another change in your local environment with an untracked file

If you want to stash these changes, you include the -u option.

```
$ git stash -u
```

Figure 11-4 shows what things look like after this command. The first element on your stash queue is moved to stash@{1}. The stashed changes from this operation are also added to the queue as the new stash@{0}. Again, your local environment is reset back to the state of the current HEAD.

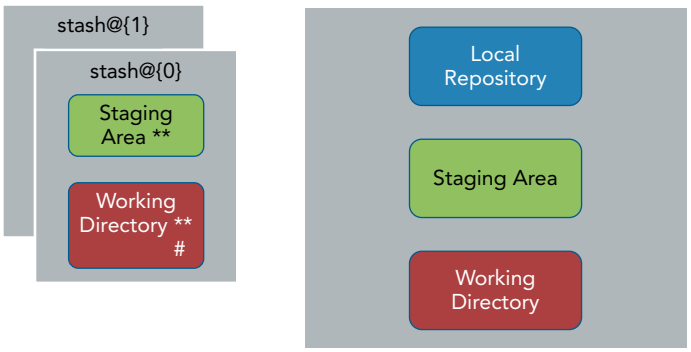


FIGURE 11-4: After stashing, including the untracked file

Stashing with a Description

Normally when Git stashes something, it has a generated comment associated with the element, of the following form:

```
stash@{#}: WIP on <branch>: <SHA1 of last commit> <last commit message>
```

Here, *WIP* stands for *work in progress*.

However, if you'd like to have a more meaningful message attached to the element, you can supply it when you do the stash. This functionality can be used to capture some quick context for future reference such as Changes for issue #123456 or Test 1 failed. Note that this is intended only as a description—visible when listing the elements of a queue—not as a handle or identifier that can be used to select or retrieve an element.

Suppose you have another change in your local environment, as shown in Figure 11-5.

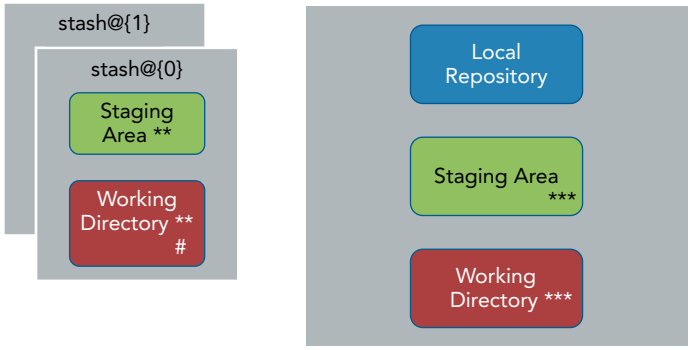


FIGURE 11-5: Another change in your local environment

To do a custom comment, you also need to specify the save command as part of the invocation string.

```
$ git stash save "changes in progress for issue #12345"
```

After this command, you have another new element on the queue and your local environment is again reset, as shown in Figure 11-6.

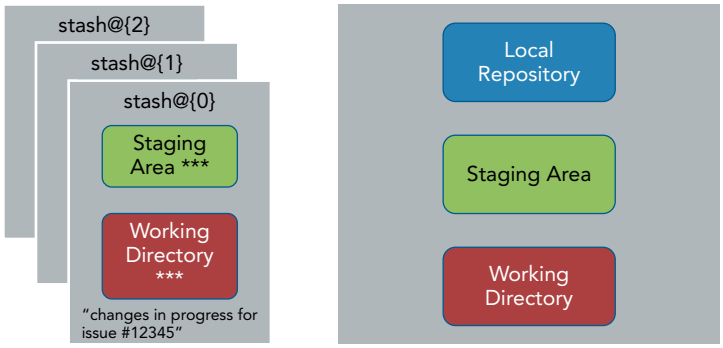


FIGURE 11-6: The third element on the queue

Seeing What's in the Stash

Once you have your changes stashed, you can look at what you have in the queue. To do this, you use the list option of the stash command. So, given the sequence you just went through, if you run the following command

```
$ git stash list
```

you see something that looks like this:

```
stash@{0}: On master: changes in progress for issue #12345
stash@{1}: WIP on master: 1d9df4f first commit
stash@{2}: WIP on master: 1d9df4f first commit
```

In the default output, I have the *name* of the stash element listed (stash@{#}), followed by information about the branch that was current when the stash was made, and then the last commit that the stash was based on.

This information is useful to a point, but what if you want to see more detail about an item in the stash? As it turns out, stash supports options like those you use with the git log command. Knowing this, you have different ways to get additional information. For example, you can start by using the --oneline option, as in

```
$ git stash list --oneline
```

which shows you the abbreviated SHA1 values for each stash element.

```
8f2728f refs/stash@{0}: On master: changes in progress for issue #12345
cc7b784 refs/stash@{1}: WIP on master: 1d9df4f first commit
12cd281 refs/stash@{2}: WIP on master: 1d9df4f first commit
```

From there, you can use the show subcommand and pass it the SHA1 value to see a quick summary of the changes that were in progress, as in

```
$ git stash show cc7b784

 file1.c | 1 +
 1 file changed, 1 insertion(+)
```

For even more information, you can add the -p (patch) option to see the patch-style differences that were in the change:

```
$ git stash show -p cc7b784

diff --git a/file1.c b/file1.c
index f2e4113..a359806 100644
--- a/file1.c
+++ b/file1.c
@@ -1 +1,2 @@
 stuff
+new
```

You now know how to stash uncommitted content and view it in the stash. Next, I discuss how to restore the content from the stash into your local environment.

Restoring Changes

When getting stored changes out of the stash, Git attempts to reapply the stashed changes from the staging area back into your local environment's staging area and the stashed changes from your working directory back into your local environment's working directory. There are two options for doing this: apply and pop. These options can be used on any branch, not just the branch that the original stash was performed on.

The Apply Option

The apply option attempts to put your changes back while leaving a copy of the items as an element still in the queue. Note that you can apply from any element at any position in the queue by referencing the name in the stash (stash@{#}). Unlike a more formal queue, you do not have to do pull elements only from the first or last positions in the queue. An example usage would be

```
$ git stash apply stash@{1}
```

If the command is successful, your staging area and working directory are updated with the contents of the element from the stash and Git runs a git status command to show you the updated status. (Note that there is also a git apply command that is used to apply patches, so be careful not to confuse these two commands.)

The Pop Option

The pop option works like the apply option, but it removes the element from the queue after updating your environment. Like the apply option, you can pop an element from any position in the queue. An example would be

```
$ git stash pop stash@{2}
```

After an apply and pop operation in my example, your local environment and queue might look like Figure 11-7.

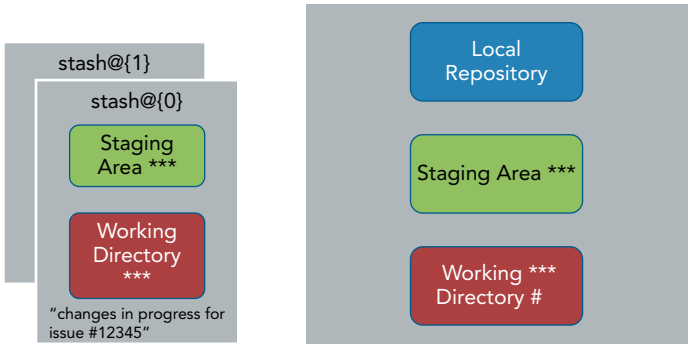


FIGURE 11-7: Queue and local environment after an apply and pop from the stash

Notice that on the queue, because you did the pop on element 2, you are now down to two elements (0 and 1). Also, in your local environment's staging area and working directory, you have the combination of apply and pop. In this situation, you didn't have any merge issues or conflicts, but that isn't always the case. Let's look at what can happen when you do.

Merge Considerations When Performing a Pop or Apply

When you're ready to bring content back from the queue into your local environment, chances are that you may have other updates in your local environment that weren't there when you originally performed the stash. If there are potential merge conflicts between what you are trying to apply or pop and what's currently in your local environment, what Git does depends on the circumstances.

As an example, if you still have untracked files in your working directory that would be overwritten by an apply or a pop, Git warns you and stops trying to populate from the queue, as shown here:

```
$ git stash apply stash@{1}

error: The following untracked working tree files would be overwritten by merge:
        file2.c
Please move or remove them before you can merge.
Aborting
```

Suppose you then stage file2.c (making it tracked) and try the stash again. This time (assuming there are no other untracked files), Git tries to merge the stash in. In this case, though, if file2.c is different, you can get merge conflicts that you have to resolve manually.

```
$ git add file2.c

$ git stash apply stash@{1}
Auto-merging file2.c
CONFLICT (add/add): Merge conflict in file2.c
```

Note that regardless of what you have staged, if you try to do a commit, Git does not allow it because there are unmerged files locally. The solution is to fix the merge issues first and then stage the fixed files.

Staging Commands

In previous chapters, you got used to the idea of staging new content or updates into the staging area before committing the new content into the local repository. As it turns out, Git employs this model not only for file content, but also for operations that manipulate the layout of content it controls.

Examples where this model would be applied are the commands to rename, move, or delete content. For each of these commands, when you execute the corresponding Git command, two things happen:

1. The change in name or structure is made in the working directory.
2. The change in name or structure is staged into the staging area.

At this point, if you do a *git status* command, you see a status corresponding to the operation being done and the files involved.

When you are ready to have the change applied to the local repository, you then do a git commit, just as you do to commit content changes. You can think of this as *committing to making the change that's staged for the local repository*. This is a slightly different way to think about using the staging area and committing operational changes. However, it makes sense in the context of the Git model if you think about it.

Take a look at the following sections to get a better idea of how this model works.

mv

The next command to consider is the *mv* command. You may have noticed that this command has the same name as the Linux operating system command *mv*. It essentially does the same thing: it allows you to rename a file or move it to a different location. The syntax is as follows:

```
git mv <options>... <args>...
```

Renaming Content

In its simplest form, the mv command is just doing a rename of a file. Git does not track metadata about renames, but instead infers this metadata via the stored SHA1 values. This means that regardless of the name or path in which a file is stored in a structure, it has the same internal SHA1 value inside the Git repository. If the destination file that you are trying to use as the new name exists, Git reports this back as an error. If that occurs and you want to rename the file anyway, you can use the -f option to force the rename. The syntax is as follows:

```
$ git mv [old filename] [new filename]
```

Suppose you issue the following command in your working directory:

```
$ git mv file1 file2
```

If you now look in that directory, you see file2 instead of file1 as expected. If you do a git status command, you see the staged operation:

```
renamed: file1 -> file2
```

The change has been made in the working directory and is staged. At this point, no changes for this operation have been made in the local repository. To complete the operation and have the change take effect in the local repository, you just need to do a commit operation:

```
$ git commit -m "renaming file1"
```

Now your file has been renamed at all of the local levels.

Moving Content

In the case of moving a file, the primary difference from renaming is that the second argument to the mv command contains a directory.

```
$ git mv [old filename] [directory[new filename]]
```

Moving is still considered a *rename* in Git, but it is renaming to a different path and not just a filename. Again, the -f option is available to force the rename if needed. As an example of this version of the command, suppose you have a subdirectory *subdir1* and you issue the following command in your working directory:

```
$ git mv file2 subdir1/file3
```

If you now look in subdir1, you see file3 there, as expected.

If you do a git status command, you see the staged operation, ready to be committed:

```
renamed: file2 -> subdir1/file3
```

Once again, the change has been made in the working directory and is staged. At this point, no changes for this operation have been made in the local repository. To complete the operation and have the change take effect in the local repository, you just need to do a commit operation.

```
$ git commit -m "moving file2 to subdir1"
```

rm

Like the mv command, the rm command has the same name as the Unix command. Also like the Unix command, the purpose of the rm command is to delete (remove) content, from Git. The syntax for the rm command is as follows:

```
git rm [-f | --force] [-n] [-r] [--cached] [--ignore-unmatch] [--quiet] [--]
<file>...
```

The rm command can accept either filenames or directory names as arguments. If directory names are passed or a pattern is used, only those files under Git control are affected.

This command also operates with the staging and commit workflow that Git uses. Like the mv command, when you do a git rm command, it makes the change locally and stages it in the staging area. An example status at this point would look like this:

```
deleted: filename
```

With the rm command, Git checks first to see if you have an updated version of content in the working directory. If you do, then it stops. The idea here is that you have local changes that you may have intended to put into Git.

If you want to override Git and remove the content, you can use the -f option to force the removal. If you have a staged version and want to just remove it from the staging area, you can use the --cached option. (Recall from my discussion on git diff that the term *cache* is another [historical] name for the staging area.) Here again, you would do a git commit to finalize this change in the local repository.

MIXING OPERATIONS AND CONTENT CHANGES

It is okay to have a mixture of updated or new files staged as well as operations staged at the same time. It all comes down to staging what you want to put in or change in the repository, and then committing to it.

COMMANDS FOR SEARCHING

In this section, I cover a couple of ways you can use commands in Git to search for things. The first way is to use a specialized command that bears the same name as an operating system command, and the second way is to use a specialized form of a command that I've already covered.

grep

The grep command in Git provides a convenient (and probably familiar) way to search for regular expressions in your local environment. The syntax for *git grep* is as follows:

```
git grep [-a | --text] [-I] [--textconv] [-i | --ignore-case] [-w | --word-
regexp]
          [-v | --invert-match] [-h|-H] [--full-name]
          [-E | --extended-regexp] [-G | --basic-regexp]
          [-P | --perl-regexp]
          [-F | --fixed-strings] [-n | --line-number]
          [-l | --files-with-matches] [-L | --files-without-match]
          [(-O | --open-files-in-pager) [<pager>]]
          [-z | --null]
          [-c | --count] [--all-match] [-q | --quiet]
          [--max-depth <depth>]            [--color[=<when>] | --no-color]
          [--break] [--heading] [-p | --show-function]
          [-A <post-context>] [-B <pre-context>] [-C <context>]
          [-W | --function-context]
          [--threads <num>]
          [-f <file>] [-e] <pattern>
          [--and|--or|--not|(|)|-e <pattern>...]
          [ [--[no-]exclude-standard] [--cached | --no-index | --untracked] |
<tree>...]
          [--] [<pathspec>...]
```

You will probably recognize some of these options as looking the same as their operating system grep command counterparts (such *as -v, -E, -F, -c,* and so on). In fact, much of the functionality of this grep command can be understood by using it the same way as the operating system grep command—with an indication of which levels in the local environment you want to search.

By default, this command searches for the requested expressions across all tracked files in the working directory. Here's an example:

```
$ git grep database -- *.java
api/src/main/java/com/demo/pipeline/status/status.java:      @Path("/database")
dataaccess/src/main/java/com/demo/dao/MyDataSource.java:
logger.log(Level.SEVERE, "Could not access database via connect string
jdbc:mysql://"+strMySQLHost+":"+strMySQLPort+"/"+strMySQLDatabase,e);
```

These options simply tells the command to search for all instances of the expression (*database*) in files with the .java extension. Notice the use of the double hyphen (--) separator here. As with other Git commands, the double hyphen separates the command from path-limiting options. The part before the double hyphen is the actual command, and the part after the double hyphen is the selected set for the command to operate against.

Git provides several useful options to provide additional information for the grep command. One is the *-p* option, which tries to show the header of the method or function where a match occurs.

```
$ git grep -p database -- *.java
api/src/main/java/com/demo/pipeline/status/V1_status.java=public class V1_status {
```

```
api/src/main/java/com/demo/pipeline/status/V1_status.java:      @Path("/database")
dataaccess/src/main/java/com/demo/dao/MyDataSource.java=public class MyDataSource {
dataaccess/src/main/java/com/demo/dao/MyDataSource.java:
logger.log(Level.SEVERE, "Could not access database via connect string
jdbc:mysql://"+strMySQLHost+":"+strMySQLPort+"/"+strMySQLDatabase,e);
```

You can also use the standard git config command to configure several options to be on by default when running the git grep command. For example, to see line numbers when running the command, you normally need to pass the -n option. However, if you want this to always be the default option, you can configure the git config setting of `git.lineNumber` to `true`. So, both of the following operations would result in the output including line numbers:

```
$ git grep -n database -- *.java
```

or

```
$ git config grep.lineNumber true

$ git grep database -- *.java
api/src/main/java/com/demo/pipeline/status/status.java:31:   @Path("/database")
dataaccess/src/main/java/com/demo/dao/MyDataSource.java:64:
logger.log(Level.SEVERE, "Could not access database via connect string
jdbc:mysql://"+strMySQLHost+":"+strMySQLPort+"/"+strMySQLDatabase,e);
```

See the help page for the Git grep command for a complete list of settings that you can configure.

To make the output easier to read, you can use the --break option, which prints a separator line between matches from different files, and the --heading option, which prints the filename as a header above the matches for a file instead of on each line. Incorporating these options back into my example with the -p (and the -n configured as a default), you get the following output, which is much easier to digest:

```
$ git grep -p --break --heading database -- *.java

api/src/main/java/com/demo/pipeline/status/status.java
13=public class V1_status {
31:     @Path("/database")

dataaccess/src/main/java/com/demo/dao/MyDataSource.java
18=public class MyDataSource {
64:                logger.log(Level.SEVERE, "Could not access database via connect
string jdbc:mysql://"+strMySQLHost+":"+strMySQLPort+"/"+strMySQLDatabase,e);
```

You can also use Boolean operators to specify how to grep for multiple instances. For example, to require that two expressions are found in the same line, you can use the --and option.

```
$ git grep -e 'database' --and -e 'access' -- *.java
dataaccess/src/main/java/com/demo/dao/MyDataSource.java:64:
logger.log(Level.SEVERE, "Could not access database via connect string
jdbc:mysql://"+strMySQLHost+":"+strMySQLPort+"/"+strMySQLDatabase,e);
```

When you do use Boolean operators, you need to meet a couple of requirements:

- The expressions must be surrounded with quotes.
- The expressions must have the -e option in front of them.

These requirements help to avoid confusion with filenames and other parts of the command.

The or option is just --or, with the same requirements and syntax as for and.

```
$ git grep -e 'database' --or -e 'access' -- *.java
api/src/main/java/com/demo/pipeline/status/status.java:31:   @Path("/database")
dataaccess/src/main/java/com/demo/dao/MyDataSource.java:64:
logger.log(Level.SEVERE, "Could not access database via connect string
jdbc:mysql://"+strMySQLHost+":"+strMySQLPort+"/"+strMySQLDatabase,e);
```

Because the --or option is the default Boolean option, the following command returns the same output as the previous one:

```
$ git grep -e 'database' -e 'access' -- *.java
```

So far, the examples you've looked at have scanned the working directory. The advantages that the git grep command has over a standard utility like grep are that it is (relatively) faster, and that it allows you to search in the standard levels of Git. For example, you can tell Git to grep in the HEAD revision or in one further back.

```
$ git grep -e 'database' HEAD  -- *.java
$ git grep -e 'database' b2e575a  -- *.java
```

There is also an option to tell Git to search in the index (staging area), --cached. Here's an example:

```
$ git add config.txt

$ rm config.txt

$ git status config.txt
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   config.txt

Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        deleted:    config.txt

$ git grep -e 'config' -- '*.txt'

$ git grep -e 'config' --cached -- '*.txt'
config.txt:1:testing config
```

As suggested in the usage information for the command, grep has a lot of other options and ways to take advantage of this search tool. See the help page for the grep command for more details.

Git Log Searches

The git log command also includes some very useful search options. When using this facility of the log command, you are primarily searching for when something was introduced or changed.

The first option is the -S option. By default, this option takes a string (not a regular expression) and searches the commit history for commits that add or delete that string. (Another way to think of this is that it searches for commits that change the number of occurrences of the string.)

The -S option is often referred to as the *pickaxe option*. There are various reasons why it has this name. For example, you can think of it as a pickaxe that's used to dig into things as you're digging into the history for a Git repository. It could also represent the shape of the -S option specification, which, if you use a little imagination, looks similar to a pickaxe.

As an example of the -S or pickaxe option, suppose that you start with the following file:

```
$ cat lines.txt
This is line 1.
This is line 2.
This is line 3.
In this line you should have line 4.
And then line 5.
```

You then make a series of changes.

```
$ git log --oneline
584832a Editorial comments
a39b87e Change wording on line 4.
f6d279d Delete line 5.
a0c6caa Initial commit of lines.txt
```

Finally, you arrive at the following version:

```
$ cat lines.txt
This is line 1.
This is line 2.
This is line 3.
Line 4 should be on this line.
This file is just a bunch of lines.
Do not add any more lines.
```

If you now use the pickaxe option to look for the string line 5, you get the following output:

```
f6d279d Delete line 5.
a0c6caa Initial commit of lines.txt
```

The reason for this is that commit a0c6caa introduced line 5, and commit f6d279d removed it. The other commits did not add or delete that string.

If you want to use the pickaxe option but supply a regular expression, you can add the -pickaxe-regex option. As the name suggests, this allows you to pass a regular expression for the argument to the pickaxe (-S) option.

```
$ git log --oneline --pickaxe-regex -S "line [1-3]"
a0c6caa Initial commit of lines.txt
```

In this case, you use the git log command to find the commits that added or deleted occurrences of line 1, line 2, and line 3.

A similar option that Git provides with the log command is -G. This option takes a regular expression and searches for commits that have added or removed lines in their patch containing the search text.

This may sound just like the definition for the -S -pickaxe-regex option. However, consider the idea that the patch for a commit can change the text surrounding the search string but not the actual search string. In that case, the number of occurrences of the search string does not change. Thus, the --pickaxe-regex -S option does not detect a change that added or deleted the search string and does not flag it, whereas the -G option does.

As an example, let's look at the wording change I made for line 4.

```
commit a39b87e2112d48f561a9042f213865a76b6c27a8
Author: Brent Laster <bcl@nclasters.org>
Date:   Tue Jul 5 15:04:26 2016 -0400

    Change wording on line 4.

diff --git a/lines.txt b/lines.txt
index ecba54a..1cd123c 100644
--- a/lines.txt
+++ b/lines.txt
@@ -1,4 +1,4 @@
 This is line 1.
 This is line 2.
 This is line 3.
-In this line we should have line 4.
+Line 4 should be on this line.
```

You have a definite change, but there are still two occurrences of "[Ll]ine 4" in the file. So, if you use the pickaxe regular expression option, you don't see this commit.

```
$ git log --oneline -S "[Ll]ine 4" --pickaxe-regex
a0c6caa Initial commit of lines.txt
```

However, if you use the -G option, you do.

```
$ git log --oneline -G "[Ll]ine 4"
a39b87e Change wording on line 4.
a0c6caa Initial commit of lines.txt
```

This is because the changes in the line do not change the number of occurrences (add to or delete from) of the search string in that line, although there is a change around the search string.

Another useful option here is the one for tracing the changes for a set of lines or a function within a file. This is invoked by the -L option and has two different forms:

-L <start>,<end>:<file>

-L :<funcname>:<file>

These forms are pretty self-explanatory. The first form takes a starting and ending line number and a file path. If you take an example from the file you've been working with to just look at lines 4 and 5, it looks like this:

```
$ git log -L4,5:lines.txt
commit 584832a9379e6f67b188a04caf2ad16d89fbeb43
Author: Brent Laster <bcl@nclasters.org>
Date:   Tue Jul 5 15:10:44 2016 -0400

    Editorial comments
```

```
diff --git a/lines.txt b/lines.txt
--- a/lines.txt
+++ b/lines.txt
@@ -4,1 +4,2 @@
 Line 4 should be on this line.
+This file is just a bunch of lines.

commit a39b87e2112d48f561a9042f213865a76b6c27a8
Author: Brent Laster <bcl@nclasters.org>
Date:   Tue Jul 5 15:04:26 2016 -0400

    Change wording on line 4.

diff --git a/lines.txt b/lines.txt
--- a/lines.txt
+++ b/lines.txt
@@ -4,1 +4,1 @@
-In this line we should have line 4.
+Line 4 should be on this line.
```

WORKING WITH PATCHES AND ARCHIVES FOR CHANGES

On many open-source projects where developers may be geographically scattered, generating patches and sharing those patches via mechanisms such as e-mail is one option for collaboration. Individual Git users are less likely to do this today, but it is still good to know how to leverage this functionality in case you need to create a patch or archive to share with someone (such as a co-worker who doesn't have Git). This section covers various commands related to generating and sharing patches, including via e-mail.

archive

The git archive command allows you to create an external packaged version of commits suitable for distribution to users or processes that may not have Git. The syntax is as follows:

```
git archive [--format=<fmt>] [--list] [--prefix=<prefix>/] [<extra>]
            [-o <file> | --output=<file>] [--worktree-attributes]
            [--remote=<repo> [--exec=<git-upload-archive>]] <tree-ish>
[<path>...]
```

A couple of simple examples here should give you a general idea of how to use this command. For example, to create a zip file, you can use this command:

```
$ git archive master --format=zip --output=../my_archive_as_of_date.zip
```

To create a tarball, you can use a command like this:

```
$ git archive master --format=tar --output=../my_archive_as_of_date.tar
```

This command creates an archive of the latest snapshot from master as a compressed file of the given format. This acts like an export command in many other source management applications, creating a compressed copy of the flat files from the repository for the given branch without the repository metadata (without the .git directory). This archive is then suitable for sharing with others, even those who don't have Git.

A few points about this command:

- Although I use master in the example, you can use any branch, HEAD, tag, or revision.
- In this case, master can be before or after the options.
- You can also specify subdirectories to archive.
- You can specify the desired level of compression by supplying a numeric option, in the range of -0 to -9, where -0 effectively says *don't compress at all* and -9 is the highest level of compression.

The following example incorporates some of these points:

```
$ git archive --format=zip mytag subdir -9 > ../myarchive.zip
```

In this example, you are archiving in a zip format, against a tag that you have put on a repository, using only the subdir subdirectory, with the highest compression level, and redirecting the output into a file rather than using the --output option.

bundle

Like the archive command, the bundle command in Git can be used to create a separate package of files associated with the repository. The primary difference here is twofold:

- The resulting bundle includes the underlying repository (the .git directory).
- The resulting bundle file can be used like a remote repository.

The primary intention for the bundle command is to allow for sharing repositories to systems where a direct connection (and thus connecting via the standard Git protocols [SSH, HTTP, and so on]) isn't possible or practical. *Bundling* allows you to store the repository into a file that you can then share to other locations in any of the typical ways (USB drive, e-mail, and so on).

The syntax for the bundle command is as follows:

```
git bundle create <file> <git-rev-list-args>
git bundle verify <file>
git bundle list-heads <file> [<refname>...]
git bundle unbundle <file> [<refname>...]
```

Note that this is another Git command that uses subcommands. Let's take a quick look at how this command can be used.

Suppose you have a local repository in directory myrepo, and you want to bundle up the master branch. (See the "filter-branch" section for information about git rev-list arguments.) You can use the following command:

```
$ git bundle create ../myrepo.bundle master
```

Then, you can transfer the bundle file to another system and clone a repository from it.

```
$ git clone -b master ../myrepo.bundle myrepo
```

> **NOTE** *I discuss the general idea of cloning in Chapter 4. As a reminder, it refers to the idea of creating a local repository from a remote repository through a* copy *process. The -b master option here refers to extracting only the one branch because that's what you bundled. I discuss cloning in detail in Chapter 12.*

The cloning operation here creates myrepo as your local repository and establishes the bundle file as your *remote repository*, just as if you had cloned from a remote repository on a server somewhere.

There is a command you can use to look at which remote repository is associated with a local repository: git remote -v. If you change into the myrepo area and run that command, you see that the bundle file is being used as your remote repository.

```
$ git remote -v
origin  C:/Users/bcl/bundle-demo/../myrepo.bundle (fetch)
origin  C:/Users/bcl/bundle-demo/../myrepo.bundle (push)
```

You can also use commands similar to the log options to store a subset of commits in the bundle. For example, the following command stores the last five commits:

```
$ git bundle create ../myrepo.bundle -5 master
```

Or, you can store the changes from the last five days:

```
$ git bundle create ../myrepo.bundle --since=5.days
```

As another example, to create an update from the last time you bundled, you can use a range starting with a relative commit from the past or with a tag that you applied in the past (perhaps the first time you did the bundle):

```
$ git bundle create ../myrepo.bundle master~5..master
```

or

```
$ git bundle create ../myrepo.bundle firstBundleTag..master
```

You can then take this incremental bundle file that you created with the recent range of commits and copy it to the same location where you put the original bundle (c:/users/bcl/bundle-demo/../myrepo .bundle). After that, because that is where the remote reference points, you can use the bundle like an updated remote repository. This means that you use commands such as git pull (which I cover in more detail in Chapter 12).

Sharing Patches through E-mail

So far in this section, I've talked about packaging up files and repositories to share with others. Since its early days, Git has provided tight integration with the simpler mailing tools and formats to share patches. (*Patches* here refer to a set of changes extracted from a Git system that can be applied to another Git system to re-create the same changes.)

Although e-mail isn't commonly used anymore to distribute patches, it can still have value in sharing patches with someone who doesn't have direct access to a repository. Git is set up to support one patch per e-mail, and it sets up the subject line in the form [PATCH #/total] plus the first line

of the commit message. The [PATCH #/total] subject line provides a way of sequencing the patches together from the e-mails.

To e-mail a patch, you need to first create a patch file. The Git command for this is *format-patch*. Its syntax is shown here:

```
git format-patch [-k] [(-o|--output-directory) <dir> | --stdout]
                 [--no-thread | --thread[=<style>]]
                 [(--attach|--inline)[=<boundary>] | --no-attach]
                 [-s | --signoff]
                 [--signature=<signature> | --no-signature]
                 [--signature-file=<file>]
                 [-n | --numbered | -N | --no-numbered]
                 [--start-number <n>] [--numbered-files]
                 [--in-reply-to=Message-Id] [--suffix=.<sfx>]
                 [--ignore-if-in-upstream]
                 [--subject-prefix=Subject-Prefix] [(--reroll-count|-v) <n>]
                 [--to=<email>] [--cc=<email>]
                 [--[no-]cover-letter] [--quiet] [--notes[=<ref>]]
                 [<common diff options>]
                 [ <since> | <revision range> ]
```

Let's look at an example of how to use this functionality. Suppose you have the following history in your Git repository:

```
$ git log --oneline
ef15dca Removing test subdir on master
f3b05f9 update test case
2f2ea1e Add in testing example files
dadd160 initial commit of simple java file
```

You want to create patch files to share for the last three commits. You use the -3 argument to tell Git to create the patch files for the last three commits, and you just use HEAD as your starting revision.

```
$ git format-patch -3 HEAD
0001-Add-in-testing-example-files.patch
0002-update-test-case.patch
0003-Removing-test-subdir-on-master.patch
```

The names that are the output of the command list the patch files that Git created for each of the respective commits.

Before I discuss e-mailing these patches, let's look at the two commands that you can use to re-create the changes in your local environment using the patch.

apply

The git apply command allows you to re-create the changes specified in a patch file to the working directory and, optionally, the staging area. It does not update the local repository. The format for the apply command is as follows:

```
git apply [--stat] [--numstat] [--summary] [--check] [--index] [--3way]
          [--apply] [--no-add] [--build-fake-ancestor=<file>] [-R | --reverse]
          [--allow-binary-replacement | --binary] [--reject] [-z]
```

```
[-p<n>] [-C<n>] [--inaccurate-eof] [--recount] [--cached]
[--ignore-space-change | --ignore-whitespace ]
[--whitespace=(nowarn|warn|fix|error|error-all)]
[--exclude=<path>] [--include=<path>] [--directory=<root>]
[--verbose] [--unsafe-paths] [<patch>...]
```

The apply command is pretty straightforward. You use the following form

```
$ git apply <name of file in patch format>
```

to make the changes just in the working directory, or

```
$ git apply --cached|--index <name of file in patch format>
```

to make the changes in the working directory and in the staging area (meaning to stage the updates). You can use either the --cached or --index option for this.

am

Here, *am* stands for *apply mailbox*, with the original idea being to apply patches that were in the legacy *mbox* format. If you take a look at the first line of one of the patch files created via format-patch, you see that Git puts a dummy line in the file to make it look like the expected format.

```
$ cat 0001-Add-in-testing-example-files.patch
From 2f2ea1e30fe4630629477338a0ab8618569f0f5e Mon Sep 17 00:00:00 2001
```

So, you can use this command to apply the patch to your environment without having to read from an actual mail source. The syntax for *am* is as follows:

```
git am [--signoff] [--keep] [--[no-]keep-cr] [--[no-]utf8]
       [--[no-]3way] [--interactive] [--committer-date-is-author-date]
       [--ignore-date] [--ignore-space-change | --ignore-whitespace]
       [--whitespace=<option>] [-C<n>] [-p<n>] [--directory=<dir>]
       [--exclude=<path>] [--include=<path>] [--reject] [-q | --quiet]
       [--[no-]scissors] [-S[<keyid>]] [--patch-format=<format>]
       [(<mbox> | <Maildir>)...]
 git am (--continue | --skip | --abort)
```

Notice in the last part of the syntax description that, like rebase or merge, am is a command that puts Git into a state. In fact, am uses similar mechanisms to rebase when you run it. If the am operation fails to complete, you need to correct the issues and use the --continue option, or either skip that part (--skip) or use --abort to stop the am operation and get out of the state. Using the basic am command is just like using the apply command:

```
$ git am <name of file in patch format>
```

The key difference or value in using am versus apply is that, assuming the patch applies cleanly, am commits the changes to the local repository. The apply command stops at the working directory or staging area (if the appropriate option is applied).

So, use am if you want to merge patches into the local repository and apply if you only want to make the changes in the working directory and, optionally, the staging area.

Mailing Patch Files

> **NOTE** *In this section, I discuss the functionality that is built into Git for e-mailing patches as text and working with them in the mbox format. If you are not using the mbox format, you may prefer to just send patch files as attachments to avoid having to save the text as a file and risk issues with formatting.*

If you want to e-mail patch files, you can use the *git send-email* command. The syntax is as follows:

```
git send-email [options] <file|directory|rev-list options>...
git send-email --dump-aliases
```

The send-email command actually has a lot of options to specify e-mail configuration values, but in most cases, you can just configure a couple of values. The values you need to configure are sendemail.from and sendemail.smtpserver.

```
$ git config --global sendemail.from  <your email address>
$ git config --global sendemail.smtpserver <your email smtp server>
```

Having specified these values, you should be ready to send the patch files.

```
$ git send-email --to bcl@nclasters.org *.patch
0001-Add-in-testing-example-files.patch
0001-Removing-test-subdir-on-master.patch
0002-update-test-case.patch
0003-Removing-test-subdir-on-master.patch
(mbox) Adding cc: Brent Laster <bcl@nclasters.org> from line 'From: Brent Laster
<bcl@nclasters.org>'

From: bcl@nclasters.org
To: bcl@nclasters.org
Subject: [PATCH 1/3] Add in testing example files
Date: Thu, 30 Jun 2016 16:22:04 -0400
Message-Id: <1467318127-10664-1-git-send-email-bcl@nclasters.org >
X-Mailer: git-send-email 2.8.1.windows.1

    The Cc list above has been expanded by additional
    addresses found in the patch commit message. By default
    send-email prompts before sending whenever this occurs.
    This behavior is controlled by the sendemail.confirm
    configuration setting.

    For additional information, run 'git send-email --help'.
    To retain the current behavior, but squelch this message,
    run 'git config --global sendemail.confirm auto'.

Send this email? ([y]es|[n]o|[q]uit|[a]ll):
```

> **NOTE** *Notice that Git prompts you to confirm that you want to send the e-mail. Git also tells you that it's adding (as cc'd) anyone else it found listed in the history for the patches. This is normal, but if you don't want to spam others that Git finds in the patches, you can suppress the cc activity with --suppress-cc=<category>. (See the help page for the send-email command for more information on these categories.) You can also change the default confirm behavior.*
>
> *Also, if Git finds an e-mail address that it doesn't recognize, it gives you the option to edit it, drop it, or quit the operation.*

The patch files arrive in the mailbox looking something like this:

```
-----Original Message-----
From: bcl@nclasters.org [mailto:bcl@nclasters.org]
Sent: Thursday, June 30, 2016 4:22 PM
To: Brent Laster <bcl@nclasters.org>
Subject: [PATCH 2/3] update test case

From: Brent Laster <bcl@nclasters.org>

---
 src/test/java/TestExample1.java | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

diff --git a/src/test/java/TestExample1.java b/src/test/java/TestExample1.java index
2aa568e..bc84162 100644
--- a/src/test/java/TestExample1.java
+++ b/src/test/java/TestExample1.java
@@ -3,7 +3,7 @@
 import org.junit.Assert;
 import org.junit.Test;

-public class TestExample {
+public class TestExample1 {

   @Test public void example1() {

--
2.8.1.windows.1
```

You can then save the text from the message and use apply or am to incorporate the patches into your Git environment, as shown in the previous section.

NOTE *If you are trying to save the text from the e-mail messages as a patch file, be aware that word-wrapping (or lack thereof) can cause your saved patch file not to apply.*

In the examples that I tried, it was necessary to modify the text, as shown in Figure 11-8. This is the text before modification.

```
+  @Test public void example1() {
+
+   Assert.assertEquals("Testing with Gradle is easy", "Testing with
+Gradle is easy");
+  }
+}
diff --git a/src/test/java/TestExample2.java b/src/test/java/TestExample2.java new file mode 100644 index 0000000..ea56c2b
--- /dev/null
+++ b/src/test/java/TestExample2.java
@@ -0,0 +1,12 @@
```

This is the text after modification.

```
+  @Test public void example1() {
+
+   Assert.assertEquals("Testing with Gradle is easy", "Testing with +Gradle is easy");
+  }
+}
diff --git a/src/test/java/TestExample2.java b/src/test/java/TestExample2.java
new file mode 100644
index 0000000..ea56c2b
--- /dev/null
+++ b/src/test/java/TestExample2.java
@@ -0,0 +1,12 @@
```

FIGURE 11-8: Changing the format of a patch received in e-mail

COMMANDS FOR CLEANING UP

As you work with Git over time, you will probably build up some unneeded and untracked content in your working directory, as well as some content that is no longer used or referenced in your local repository.

While there is no specific requirement that you clean up these areas, Git provides two commands to help with cleanup: *clean* for the working directory and *gc* (garbage collection) for the local repository.

clean

You can use the git clean command to clean up untracked files from your working directory. The syntax for this command is as follows:

```
git clean [-d] [-f] [-i] [-n] [-q] [-e <pattern>] [-x | -X] [--] <path>...
```

One of the first things to note about this command is that it tries very hard to keep you from accidentally deleting things you didn't really want to delete. If you just run the command without any arguments, you see the following:

```
$ git clean
fatal: clean.requireForce defaults to true and neither -i, -n, nor -f given;
refusing to clean
```

So, Git refuses to clean the working directory even though you told it to do so. Because Git considers this command destructive and there isn't a good way to recover from it, Git requires additional options or configuration that indicate you really want to do this. One way that Git controls this is through the configuration setting for clean.requireForce. If you haven't specifically set this value to false, then it defaults to true and, as the name implies, it requires the force option for the clean operation to proceed.

Suppose you have a couple of untracked files in your working directory. An example of clean with the force option to remove these files would be as follows:

```
$ git clean -f
Removing untracked1.txt
Removing untracked2.txt
```

If you set the configuration value to false, you are telling Git not to require the force option for the command to proceed. You do this with the following command:

```
$ git config --global clean.requireForce false
```

Now, if you execute a similar command, you do not need the *-f* option.

```
$ git clean
Removing untracked1.txt
Removing untracked2.txt
```

> **NOTE** *In general, the git stash command (discussed earlier in this chapter) is considered a safer command to use. This command saves the state of any work since the last commit into a stash. The end result can be essentially the same, but the stash option allows you to easily restore items if needed, as opposed to the clean command, which automatically deletes them.*

Notice that you can also clean a particular path or file, as follows:

```
$ git clean untracked1.txt
Removing untracked1.txt
```

Now, suppose you have a subdirectory named *sub* that contains two files—ignored1.txt and ignored2.txt—and you have also created a .gitignore file that contains *sub* as a subdirectory to ignore. The help page for the clean command tells you that the -d option should be used to remove untracked subdirectories. Let's try that option:

```
$ git clean -d

$ git clean -d sub
```

The lack of output in both cases indicates that nothing was cleaned. That is because this subdirectory was listed as something for Git to ignore in the associated .gitignore file.

The clean command provides two options for working with the .gitignore file: *-x* and *-X*. Both of these options tell clean to also clean the items in the .gitignore file. This can be useful, for example, to clean out build or other output.

The difference between *-x* and *-X* is that *-x* allows for cleaning out everything, including what's ignored, and *-X* only allows for cleaning out what's ignored. To illustrate this, consider running these commands on my earlier directory example. This example has two untracked files (untracked1.txt and untracked2.txt) in the main directory, and two files (ignored1.txt and ignored2 .txt) in the subdirectory (sub). The following output shows the results of running the clean command with the two different options:

```
$ cat .gitignore
sub

$ git clean -X -d
Removing sub/

$ git clean -x -d
Removing sub/
Removing untracked1.txt
Removing untracked2.txt
```

Notice that in the first command (with -X), only the contents noted in the .gitignore file are cleaned. In the second command, the other eligible files are also removed, in addition to the content indicated by the .gitignore file. Without the -x or -X, the content indicated by the .gitignore file is ignored.

The clean command also has an interactive interface, similar to the interactive add and commit commands I discuss in Chapter 5. To get into that mode, you can add the -i (interactive) option when you invoke the clean command.

```
$ git clean --interactive -x -d
Would remove the following items:
  sub/           untracked1.txt  untracked2.txt
*** Commands ***
    1: clean                2: filter by pattern    3: select by numbers
    4: ask each             5: quit                 6: help
What now>
```

Very briefly, here's what each of the first four options does:

- clean—executes the clean command to do the actual cleaning
- filter by pattern—prompts the user for patterns to exclude from deletion

- select by numbers—provides a list of the items to be deleted, identified by unique numbers, and allows the user to specify items by entering the desired number
- ask each—interactively prompts whether to delete specific items

Other options for the clean command include -e <pattern> to add additional patterns to exclude from deletion, and -n for a dry run to show what would be done without actually making changes.

gc

One other command for cleaning up Git content is git gc. Here, *gc* stands for *garbage collection* and, unlike the clean command, which cleans up files in your working directory, gc cleans up internal content in the repository. It has two main functions:

- To compress revisions of files to save space and increase performance
- To remove objects that are no longer reachable by any connection in Git

The syntax for gc is as follows:

```
git gc [--aggressive] [--auto] [--quiet] [--prune=<date> | --no-prune] [--force]
```

I'll briefly cover a few of the options here:

- *--aggressive*—tells gc to do a more thorough job of cleaning up and optimizing the repository. This option usually requires significantly longer for the process to run.
- *--auto*—checks to see if anything needs to be done, in terms of cleaning up the repository. If nothing is to be done, it just exits. Some Git commands may run this option automatically.
- *--quiet*—executes with less output.
- *--prune* | *--no-prune*—specifies whether or not to prune loose objects in the repository that are older than the default timeframe of two weeks.
- *--force*—starts up another gc operation, even if one is already running

For the best disk space use and overall performance, you should run git gc on a regular basis. Note that, as I have mentioned, Git tries very hard not to lose any commits or otherwise clean them up. So, any references that remain to an object through indexes, branches, reflogs, and so on, will keep an item from being cleaned. There are various configuration values (listed in the gc help page) that you can set to tell Git when something should expire or be redone.

notes

The notes command can be very useful, but it doesn't fit into any of the other groupings, so I include it separately here.

At some point after making a commit, you may decide that there are additional comments or other non-code information that you'd like to add with the commit. If this is the most recent commit, you can use the git amend functionality and update the commit message. Or, if this is later in the cycle,

you can use the interactive rebase functionality that I described in Chapter 9 and use the *reword* option. Both of these functionalities have the downside of changing the SHA1 hash of the commit. For a recent change that has not been pushed to the remote repository, that may be acceptable. However, as I've consistently warned throughout this book, changing the SHA1 values of something that is already pushed to the remote repository where others may be consuming it is problematic. Also, this could potentially be a high cost for just updating comments or messaging around the change.

This is where git notes comes in. The notes command allows you to add notes to existing commits in Git. The syntax is as follows:

```
git notes [list [<object>]]
git notes add [-f] [--allow-empty] [-F <file> | -m <msg> | (-c | -C) <object>]
[<object>]
git notes copy [-f] ( --stdin | <from-object> <to-object> )
git notes append [--allow-empty] [-F <file> | -m <msg> | (-c | -C) <object>]
[<object>]
git notes edit [--allow-empty] [<object>]
git notes show [<object>]
git notes merge [-v | -q] [-s <strategy> ] <notes-ref>
git notes merge --commit [-v | -q]
git notes merge --abort [-v | -q]
git notes remove [--ignore-missing] [--stdin] [<object>...]
git notes prune [-n | -v]
git notes get-ref
```

Here's a quick example. Suppose you have a set of commits in your repository like the following:

```
$ git log --oneline
ef15dca Removing test subdir on master
f3b05f9 update test case
2f2ea1e Add in testing example files
dadd160 initial commit of simple java file
```

You can add a note to any of the existing commits by passing their SHA1 value as the reference at the end.

```
$ git notes add -m "This is an example of a note" 2f2ea1e
```

Notice that you can pass the -m option to provide a note, just as you can use that option to pass a commit message. If the -m option isn't passed, Git starts up the default editor, as it does if no -m option is passed to a commit.

If you want to look at a specific note, you can use the show subcommand.

```
$ git notes show 2f2ea1e
This is an example of a note
```

With an additional option, you can create notes in a custom namespace. For example, if you want to create a notes namespace for reviews of previous commits, you can supply that namespace to the --ref option.

```
$ git notes --ref=review add -m "Looks ok to me" f3b05f9
```

The simplest way to see notes is to use git log. By default, this command shows notes in the default namespace. To see other namespaces, you can use the --show-notes=<namespace> or --show-notes=* option to see all of them.

```
$ git log --show-notes=*
commit 80e224b24e834aaa8915e3113ec4fc635b060771
Author: Brent Laster <bcl@nclasters.org>
Date:   Fri Jul 1 13:01:58 2016 -0400

commit ef15dca5c6577d077e38a05b80670024e1d92c0a
Author: unknown <bcl@nclasters.org>
Date:   Fri Apr 24 12:32:50 2015 -0400

    Removing test subdir on master

commit f3b05f9c807e197496ed5d7cd25bb6f3003e8d35
Author: Brent Laster <bcl@nclasters.org>
Date:   Sat Apr 11 19:56:39 2015 -0400

    update test case

Notes (review):
    Looks ok to me

commit 2f2ea1e30fe4630629477338a0ab8618569f0f5e
Author: Brent Laster <bcl@nclasters.org>
Date:   Sat Apr 11 17:34:57 2015 -0400

    Add in testing example files

Notes:
    This is an example of a note

commit dadd160cf432df7a8db454bcc0eeb22988615ed9
Author: Brent Laster <bcl@nclasters.org>
Date:   Sat Jan 5 22:45:26 2013 -0500

    initial commit of simple java file
```

ADVANCED TOPICS

In the advanced topics for this chapter, I include an eclectic mix of specialized Git commands. These are not commands you would use every day (or even every month), but you will likely need each of them at some point. The three commands I'll cover are filter-branch, bisect, and rerere.

- *filter-branch* is for splitting or tweaking repositories (sometimes using another command, *rev-list*).
- *bisect* is for quickly finding where a problem or change was introduced.
- *rerere* is for teaching Git how to automatically resolve custom merge situations and having it remember and execute the solution if the situation is encountered again.

filter-branch

In most cases throughout this book, you've been dealing with user-friendly porcelain commands that are focused on typical user operations rather than utility commands. However, one of the plumbing commands can prove very useful and is worth covering at a high level with some simple examples: filter-branch.

The syntax for filter-branch is as follows:

```
git filter-branch [--env-filter <command>] [--tree-filter <command>]
       [--index-filter <command>] [--parent-filter <command>]
       [--msg-filter <command>] [--commit-filter <command>]
       [--tag-name-filter <command>] [--subdirectory-filter <directory>]
       [--prune-empty]
       [--original <namespace>] [-d <directory>] [-f | --force]
       [--] [<rev-list options>...]
```

There are a couple of things worth pointing out about the syntax. First, notice that a number of different types of filters can be supplied to the command. You can think of these filters as being like domains—that is, operating against messages, commits, subdirectories, and so on. In some cases, this is the area the command works against, and in other cases, it's the area the command uses to do its work.

Notice that most of these filters take an additional <*command*>. The idea here is that filter-branch is traversing the commits and file hierarchies for the specified domains and executing the specified commands against those objects in the domains as it goes.

The exception to this flow for a filter is the subdirectory filter, which takes a subdirectory as an argument instead of a command. This filter is used primarily to split subdirectories out into separate repositories.

rev-list

Notice the <rev-list options> in the last line on the command line in the previous section. Git *rev-list* is another plumbing command. Its main use is to list a set of commits bounded by some range or criteria. It's pretty straightforward to use, so I'll just share the syntax here. You can use the rev-list help page to find out more details on specifying ranges if you're interested.

```
git rev-list [ --max-count=<number> ]
             [ --skip=<number> ]
             [ --max-age=<timestamp> ]
             . . . (lots of other omitted options)
             [ --count ]
             [ --use-bitmap-index ]
             <commit>... [ -- <paths>...
```

So what does rev-list have to do with filter-branch? Rev-list options can be used to bound the set of things for filter-branch to operate against. In most cases, if you need to supply an option for rev-list, you can just use the --all option to indicate all commits.

Note, however, that when using filter-branch, you have to use the revision separator, the double hyphen (--), before options intended specifically for rev-list. This is to avoid confusion with options intended specifically for filter-branch.

> **WARNING** *The standard warning about changing things already pushed to the remote repository applies here as well. If you plan to modify a local repository using this command that will be pushed back to an existing remote repository, make sure to coordinate with anyone else who may already be using a pre-filter-branch version of the remote repository.*

Precautions

If the following section is not clear, refer to Chapter 12, where I discuss remotes.

When pushing the results of a filter branch to a new destination remote repository, take care not to use the existing remote name in the push command and overwrite the old repository (unless that's your intent). Some people recommend removing the reference to the old remote to avoid accidentally overwriting things. However, if you do that, and you have remote tracking branches that you have not branched locally (that is, origin/foo versus foo), you will lose those references when you remove the remote reference. In those cases, it's a good idea to create the local branch first via git branch <foo> before removing the reference.

To remove the reference, assuming you have a remote named origin that can be disconnected, you can use the command - *git remote rm origin*.

Also, if you have other changes in the repository, you may want to make a copy of the repository before running a filter-branch. I talk later about how to undo a filter branch.

Examples

Let's take a look at a couple of examples of how you can use filter-branch. Note that while these are simple examples, it is possible to do much more complicated things because you can pass any executable script, program, or command as a <command> (for the filters that take <command> arguments).

Example 1: Splitting a Subdirectory Out into a Separate Repository

In the first example, you'll use the subdirectory filter. This filter does not take a command but rather a subdirectory. The most common use for this filter is to split out an entire subdirectory into a separate repository. For that, the syntax is straightforward.

Suppose you have created or cloned a structure into a local repository that looks like this:

```
$ ls
agents.sql           db.properties            sonar-project.properties
api/                 debug.save               tree.out
build.gradle         gradle.properties        util/
build.gradle.jetty   gradle.properties.admin  web/
build.gradle.tomcat  gradle.properties.safe
dataaccess/          settings.gradle
```

You are interested in splitting the web subdirectory tree out into its own repository. First, you note what the structure looks like under web, as well as the log, for a reference point.

```
$ ls -la web
total 36
drwxr-xr-x 1 bcl 1049089     0 Jun 16 14:25 ./
drwxr-xr-x 1 bcl 1049089     0 Jun 16 18:21 ../
-rw-r--r-- 1 bcl 1049089 18905 Jun 16 14:25 .classpath
-rw-r--r-- 1 bcl 1049089  1112 Jun 16 14:25 .project
drwxr-xr-x 1 bcl 1049089     0 Jun 16 14:25 .settings/
drwxr-xr-x 1 bcl 1049089     0 Jun 16 14:25 src/


$ git log --oneline
7ae0228 adding some initial testing examples
4758f9e fix conflicts
d8a1065 adding updates
f64bd2c update for latest changes to make web server generic and compatible with
tomcat
0c191a4 update for retrieving artifact from artifactory
1e8173e add in sample tests and jacoco configuration
c228ad7 Update gradle and remove tmp files
82b07c9 Adding sonar properties file
845bf97 update script
4a4fe0e updated to publish to artifactory
b2e575a sql to recreate db
c6b5cbd update for db.properties
be42303 initial add of files from gradle build

$ git log --oneline web
4758f9e fix conflicts
d8a1065 adding updates
f64bd2c update for latest changes to make web server generic and compatible with
tomcat
c228ad7 Update gradle and remove tmp files
c6b5cbd update for db.properties
be42303 initial add of files from gradle build
```

Now, you can run the actual filter-branch command.

```
$ git filter-branch -f --subdirectory-filter web -- --all
Rewrite be42303ffb9356b8e27804ce3762afdeea624c64 (1/6) (0 seconds passed,
remainRewrite c6b5cbd8805bc7b1b411a89be66adccc037df553 (2/6) (1 seconds passed,
remainRewrite f64bd2ce520e9a3df4259152a139d259a763bc31 (2/6) (1 seconds passed,
remainRewrite d8a1065e9709d2c5ee20f62fd4e338fe35666c65 (2/6) (1 seconds passed,
remainRewrite c228ad75ef060707ef2905f8bd46012ed67e718b (5/6) (5 seconds passed,
remainRewrite 4758f9e7a9cbeb8dfea0a37a416b46d823ffa95a (5/6) (5 seconds passed,
remaining 1 predicted)
Ref 'refs/heads/master' was rewritten
```

Notice the messages about rewriting SHA1 values. This is Git performing the operations it needs to in the SHA1s in the repository that have to do with the subdirectory web. At the end is the message that refs/heads/master was rewritten. This is what you're looking for to indicate something was done. If it wasn't rewritten, that would mean it was still pointing to the old structure. You can now do an *ls* command and a *git log* command to see what the repository looks like.

```
$ ls -la
total 100
drwxr-xr-x 1 bcl 1049089     0 Jun 16 18:26 ./
drwxr-xr-x 1 bcl 1049089     0 Jun 16 13:20 ../
-rw-r--r-- 1 bcl 1049089 18905 Jun 16 18:26 .classpath
drwxr-xr-x 1 bcl 1049089     0 Jun 16 18:26 .git/
-rw-r--r-- 1 bcl 1049089  1112 Jun 16 18:26 .project
drwxr-xr-x 1 bcl 1049089     0 Jun 16 18:26 .settings/
drwxr-xr-x 1 bcl 1049089     0 Jun 16 18:26 src/


$ git log --oneline
88975c2 fix conflicts
30bb02b adding updates
80233a6 update for latest changes to make web server generic and compatible with
tomcat
eea95f7 Update gradle and remove tmp files
5172f34 update for db.properties
e5a264f initial add of files from gradle build
```

Notice that your repository now consists of only the web pieces, as intended.

Undoing a filter-branch

While the current HEAD of this repository now only points to the web subtree, the commits that make up the remaining structure are still there—just not accessible in this chain. This is one of the reasons that a filter-branch to split or remove content doesn't necessarily make the repository any smaller. (The online help page for filter-branch discusses the remaining steps that you have to follow to actually remove content and shrink the repository.)

However, this is good news for you because it means that, as with nearly any other repository modification, you can undo the operation by just using a reset to a point before the operation. Git does keep a backup of the previous reference pointed to by HEAD before the filter-branch. Assuming you were working in the master branch, that SHA1 value can be found in the following path:

```
$ cat .git/refs/original/refs/heads/master
7ae022891d25aa46b86cf8df3c66204f0272f447
```

So, you can simply run the command, git reset --hard 7ae02289. Or, you can use the git reflog command to find the relative reference before the operation and then run the command, git reset --hard HEAD@{<relative offset>}.

Example 2: Deleting a File from All of the History

Suppose that you realize a file with sensitive data was committed into a repository weeks or months ago. A git rm command will remove the current instance of the file, but the previous versions will still be in the repository.

To remove a file from the history, you can use either the tree-filter or the index-filter option. Because the index-filter option can use the index instead of the working tree (think removing in the staging area versus checking out and removing), this option will be faster. Like most other filters, the index-filter option takes a command as an argument to apply when it encounters an instance of the thing

you are trying to filter out. For your purposes, you can use the git rm command here to simplify the task. You add the --cached option because you're working in the index.

There's one additional problem you can run into, though. Stuck within the help page for filter-branch is this line: "If any evaluation of <command> returns a non-zero exit status, the whole operation will be aborted." This means that if you are trying to apply a command like git rm to a commit that doesn't have it, the non-zero status for that commit will cause the operation to abort. So, to ensure that doesn't happen, you need to add the --ignore-unmatched option to the git rm command. This tells the command to exit with a zero return code even if none of the files match. So, your command to remove a file becomes

```
$ git filter-branch --index-filter 'git rm --cached --ignore-unmatch <relative path
to file>' <branch name>
```

Suppose you had a branch in your repository where a file with a temporary password named tmp_pass.txt had inadvertently been introduced to the repository at some point in the past. This was done on a branch named fix_secure before that branch was merged into master. This is discovered late in the cycle and you need to remove it from the repository.

To handle this situation, you need to run a similar command for each branch.

```
$ git filter-branch -f --index-filter 'git rm --cached --ignore-unmatch tmp_pass
.txt' fix_secure
Rewrite 50789669c4d4a4609ce06840f354eb6119884843 (1/6) (0 seconds passed,
remainRewrite 298f4ac9c3a56b1e7e85a0af3fb9ba6738f4c4c2 (2/6) (1 seconds passed,
remainRewrite 2c3a1194fb5fe190c1f81bb6e807e83be2732146 (2/6) (1 seconds passed,
remainRewrite a9528538a9dc6bc5f2955f0ed556fe2d7b0c8f79 (2/6) (1 seconds passed,
remainRewrite b8d05eea4509d98d39729c682a891da037c861ee (5/6) (4 seconds passed,
remaining 0 predicted)    rm 'tmp_pass.txt'
Rewrite 6ca7f0a69701bdb0412e5840676ad0b4c682eb85 (5/6) (4 seconds passed, remaining
0 predicted)
Ref 'refs/heads/fix_secure' was rewritten
```

In this second command, you add the -f parameter for filter-branch. This is because there is already an internal backup file that says what the branch pointed to before you ran this command. Git is storing that backup from the last run. Git only stores one backup reference, so it needs to be told that it's okay to overwrite that reference.

```
$ git filter-branch -f --index-filter 'git rm --cached --ignore-unmatch tmp_pass
.txt' HEAD
Rewrite 50789669c4d4a4609ce06840f354eb6119884843 (1/8) (0 seconds passed,
remainRewrite 298f4ac9c3a56b1e7e85a0af3fb9ba6738f4c4c2 (2/8) (1 seconds passed,
remainRewrite 2c3a1194fb5fe190c1f81bb6e807e83be2732146 (2/8) (1 seconds passed,
remainRewrite a9528538a9dc6bc5f2955f0ed556fe2d7b0c8f79 (2/8) (1 seconds passed,
remainRewrite b8d05eea4509d98d39729c682a891da037c861ee (5/8) (4 seconds passed,
remaining 2 predicted)    rm 'tmp_pass.txt'
Rewrite 295070e1e2e7748b4f1c753d6d16b3de4c25fc55 (5/8) (4 seconds passed, remaining
2 predicted)    rm 'tmp_pass.txt'
Rewrite bbe20a408cc413a949e6eaef2d9631fd7c81b22b (7/8) (6 seconds passed, remaining
0 predicted)    rm 'tmp_pass.txt'
Rewrite e36eb926872bbf54d58d8ea0934f684771a30b3a (7/8) (6 seconds passed, remaining
0 predicted)
Ref 'refs/heads/master' was rewritten
```

In the second example, you can use HEAD because you were currently on master.

Let's look at one more use case. Suppose that instead of just being in the current directory, the tmp_pass.txt file was recently moved into a temp directory. The approach is similar for this situation, except that, on each branch where this situation exists, you need to change the path to be the directory name and add the -r option to the command.

```
$ git filter-branch -f --index-filter 'git rm -r --cached --ignore-unmatch temp'
fix_secure
Rewrite 50789669c4d4a4609ce06840f354eb6119884843 (1/6) (0 seconds passed,
remainRewrite 298f4ac9c3a56b1e7e85a0af3fb9ba6738f4c4c2 (2/6) (1 seconds passed,
remainRewrite 2c3a1194fb5fe190c1f81bb6e807e83be2732146 (2/6) (1 seconds passed,
remainRewrite a9528538a9dc6bc5f2955f0ed556fe2d7b0c8f79 (2/6) (1 seconds passed,
remainRewrite 887f5b38c4aa7831feb7af1b5d122df686aa26f6 (5/6) (4 seconds passed,
remainRewrite a9b79bbf6951abcd89130ef578cbb6fc31c65cfd (5/6) (4 seconds passed,
remaining 0 predicted)    rm 'temp/tmp_pass.txt'

Ref 'refs/heads/fix_secure' was rewritten
```

NOTE *If you want to verify whether a current file was previously renamed or moved from another location, you can use the --follow option for git log. Here's an example that uses this option to show the history information for a file in its current location and with its previous location and name:*

```
$ git log --follow --name-only temp/tmp_pass.txt
commit e36eb926872bbf54d58d8ea0934f684771a30b3a
Author: Brent Laster <bcl@nclasters.org>
Date:   Fri Jun 17 12:04:28 2016 -0400

    update

temp/tmp_pass.txt

commit b8d05eea4509d98d39729c682a891da037c861ee
Author: Brent Laster <bcl@nclasters.org>
Date:   Fri Jun 17 11:54:12 2016 -0400

    needed to fix another place

tmp_pass.txt
```

Here are a few things to consider when removing files:

- The filter-branch's rm command needs to specify the file or directory path to the item you want to remove. This may involve multiple paths if the file has been moved or renamed.
- The filter-branch needs to be run against each branch where you want to do the removal.
- The filter-branch will rewrite affected SHA1s, so the SHA1 values in the chain will be different.

- The original SHA1s are still there for the time being, so you can reset back to them to undo the filter-branch.
- Because the original SHA1s are still there, this will not immediately shrink the size of the repository. The help page for filter-branch contains more information about what has to be done to shrink the repository.

Example 3: Changing the E-mail of a Set of Commits

The last example shows how to change the e-mail for a set of commits. This example illustrates using another filter, the environment filter, as well as only operating against a subset of commits in the current branch.

Suppose that you want to change the last three commits in your history to indicate that they were done at your business instead of on your personal system. Currently, your history looks like this (using a formatted log output to show the e-mail):

```
$ git log --pretty=format:"%h %ae | %s %d"
f713a23 bcl@nclasters.org | update  (HEAD -> master, refs/original/refs/heads/
master)
1d60e15 bcl@nclasters.org | prepare for release
11e90d5 bcl@nclasters.org | create license file
6aa64e2 bcl@nclasters.org | needed to fix another place
fe41715 bcl@nclasters.org | fix for secure
b7b59f2 bcl@nclasters.org | update for support code
7e0e17f bcl@nclasters.org | update core
e1214d9 bcl@nclasters.org | add initial files
```

To change the e-mail portion here, you can use the environment filter for filter-branch. Environment variables can then be used and set by the commands that you pass in. This updated environment is then used in the course of redoing the chosen commits.

In this case, you update the author e-mail by setting the environment variable GIT_AUTHOR_EMAIL. One key point here is that you have to be sure to export the new value after you modify it so that it will be set for the commit operation that is updating things. Your command looks like this if you only want to change the last three commits:

```
$ git filter-branch -f --env-filter 'GIT_AUTHOR_EMAIL=bcl@mycompany.com; export
GIT_AUTHOR_EMAIL' -- HEAD~3..HEAD
Rewrite 11e90d549c65da2a2c60d790f87bd1ddc1831dfa (1/3) (0 seconds passed, remaining
0 predicted)
Rewrite 1d60e1557b33ea181066df85f3f6b9e633d9e325 (2/3) (1 seconds passed, remaining
0 predicted)
Rewrite f713a23ae271e9f261f4dc25ef15a00d95c9ee41 (2/3) (1 seconds passed, remaining
0 predicted)
Ref 'refs/heads/master' was rewritten
```

Notice that you pass the -f option to override the last backup again, and for the rev-list range option you are passing a range from the oldest to the newest for what you want to change. The first value in the range is not changed. Read this as “Everything after HEAD~3 up to the current HEAD”, which would translate into HEAD~2, HEAD~1, and HEAD.

If you look at the log before you did the filter-branch, you can see that the SHA1 values noted as *Rewrite* are the most recent three values in your log. And if you look at the log after the filter-branch (the following code), you can see that the most recent three entries in your log have different SHA1 values since they were rewritten (and have the updated e-mail), while the remaining entries in your log have the same SHA1 values as before.

```
$ git log --pretty=format:"%h %ae | %s %d"
2b92751 bcl@mycompany.com | update  (HEAD -> master)
bd56c6a bcl@mycompany.com | prepare for release
bb0bbce bcl@mycompany.com | create license file
6aa64e2 bcl@nclasters.org | needed to fix another place
fe41715 bcl@nclasters.org | fix for secure
b7b59f2 bcl@nclasters.org | update for support code
7e0e17f bcl@nclasters.org | update core
e1214d9 bcl@nclasters.org | add initial files
```

bisect

Another useful tool that Git provides is bisect. The bisect command provides a mechanism for quickly locating where a problem or change was introduced in a range of commits.

The bisect command effectively implements an automated binary search across commits in Git. Starting from an initial range specified between a known bad revision and a known good revision, Git proceeds to keep dividing the remaining revisions roughly in half each time until the user can narrow down that a particular revision was the first time a change was introduced. Most commonly, this is used for debugging a problem—for example, when a bug first started occurring—so that developers can hone in on what code change caused the issue.

The bisect command is a git command that is really an application in itself. It takes a series of sub-commands as arguments to initiate, end, and manage the process. The syntax looks like this:

```
git bisect start [--term-{old,good}=<term> --term-{new,bad}=<term>]
         [--no-checkout] [<bad> [<good>...]] [--] [<paths>...]
git bisect (bad|new) [<rev>]
git bisect (good|old) [<rev>...]
git bisect terms [--term-good | --term-bad]
git bisect skip [(<rev>|<range>)...]
git bisect reset [<commit>]
git bisect visualize
git bisect replay <logfile>
git bisect log
git bisect run <cmd>...
git bisect help
```

I won't dive into all of these subcommands, but I will touch on the core components needed to use the command.

It's also useful to understand, when starting out, that bisect is another one of the Git state commands, meaning that once you start this command, you are in the bisecting state in Git until you complete the workflow or end it. This is similar to other state commands that I have discussed, such as merging, rebasing, and cherry-picking. Unlike those commands, though, bisect does not have an --abort subcommand. To end a bisect, you do a reset, as follows:

```
$ git bisect reset
```

or

```
$ git bisect reset HEAD
```

There are two ways to get started with bisect. One way is to start with a current revision that is known to be bad, then start the operation, check out a known previous good revision to identify the starting range, and go from there. A second way to start the operation is to pass a starting and ending range to the command invocation, such as this:

```
$ git bisect start HEAD  HEAD~10
```

Here, *HEAD* represents a known bad revision and *HEAD~10* represents a known good revision—ten commits before current HEAD. This establishes your starting range and starts the operation in one command.

Once the starting range is specified, the *bisect good* or *bisect bad* subcommand is used to indicate the state of the currently selected commit as the process continues. Git relies on the user to tell it whether the current commit it selected is good or bad. Armed with that information, Git then selects a commit roughly in the middle of the remaining range of candidates and makes it available for the user to test next. This process continues until the remaining range becomes small enough that Git and the user can zero in on the first commit where the problem occurred.

Let's look at an example. Figure 11-9 shows a series of commits in the local repository and a working directory. You are trying to figure out where a problem was introduced in the code. First, you check out the current version—version 10—and try it. As suggested by the X in the figure, this version doesn't work.

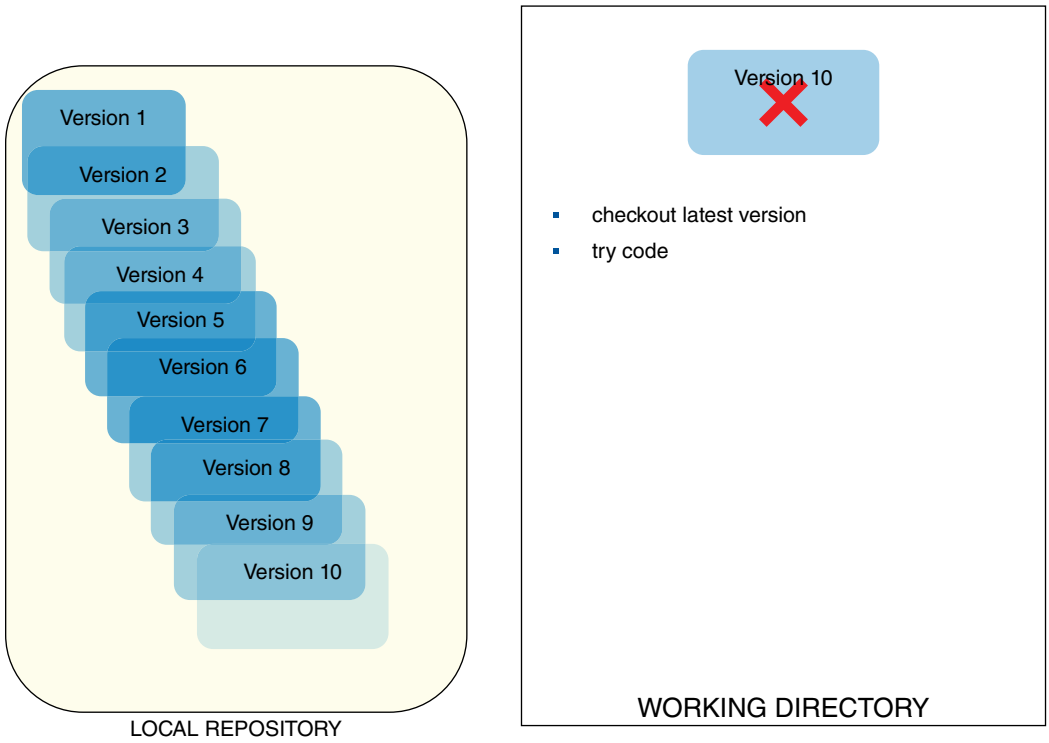


FIGURE 11-9: Starting state for bisect

From there, you can begin the bisect process to find the first bad revision. You can do that the simple way using

```
$ git bisect start
```

Then, you indicate that the current revision is bad:

```
$ git bisect bad
```

This tells Git to note this revision as bad, as indicated by the X next to the block in the repository.

Now, you need to identify a revision that is working, for the beginning of the range. To do this, you check out an earlier version. You can supply the version using a reference relative to HEAD or an explicit SHA1 from the log. You use the relative reference here.

```
$ git checkout HEAD~10
```

This puts version 1 in the working directory, so now you can try that code. In this case, version 1 works.

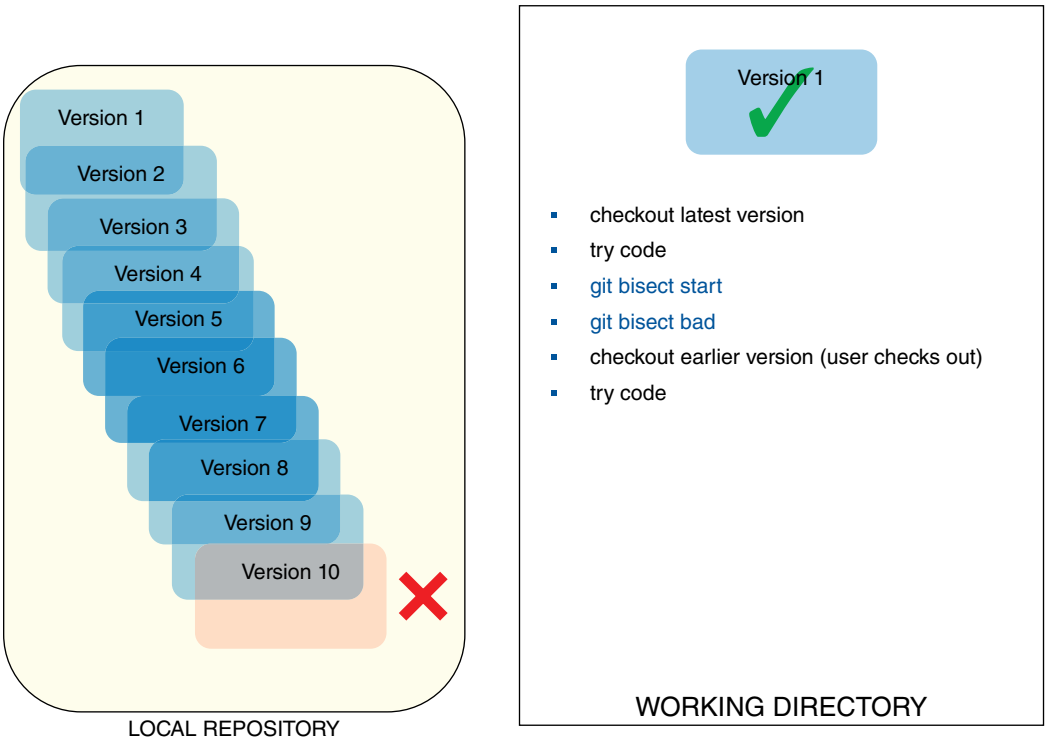


FIGURE 11-10: Checking for a good version

Because that version works, you can tell the bisect command that the revision is good. This causes bisect to mark that revision as good and then return a commit roughly halfway in between the good and bad revisions to try and see if it works. In this case, that's version 5. You can then try version 5 and see if it works. It does.

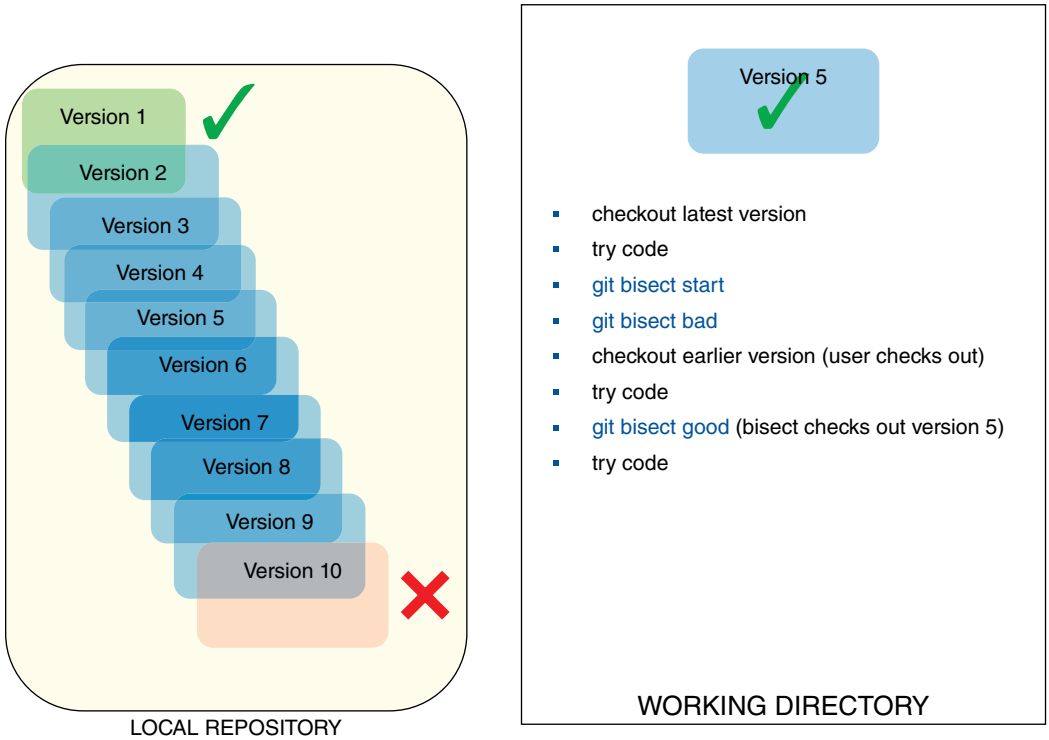


FIGURE 11-11: Initial bisect trial

You can then mark version 5 as good.

```
$ git bisect good
```

Since version 5 is marked as good, this also marks all the versions from 1 to 5 as good. This means that the first bad revision must be in between version 5 and version 10. The bisect command then gives back version 7 and the process continues.

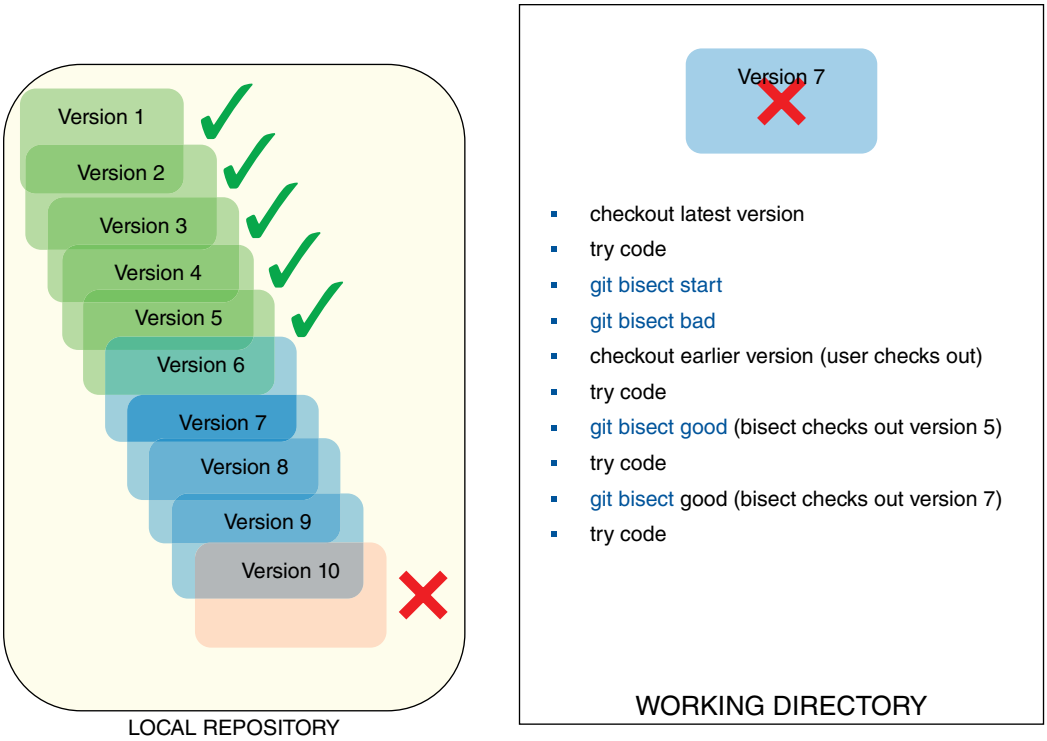


FIGURE 11-12: Bisecting—the next steps

You indicate that version 7 is bad.

```
$ git bisect bad
```

Now, you know that versions 1 to 5 are good as indicated by the checkmarks. You also know that versions 7 to 10 are bad, as indicated by the X's. You have one more version to evaluate—version 6—to determine whether the first bad version is version 6 or version 7. Git checks out the remaining version, version 6. You try it to detect whether it is bad.

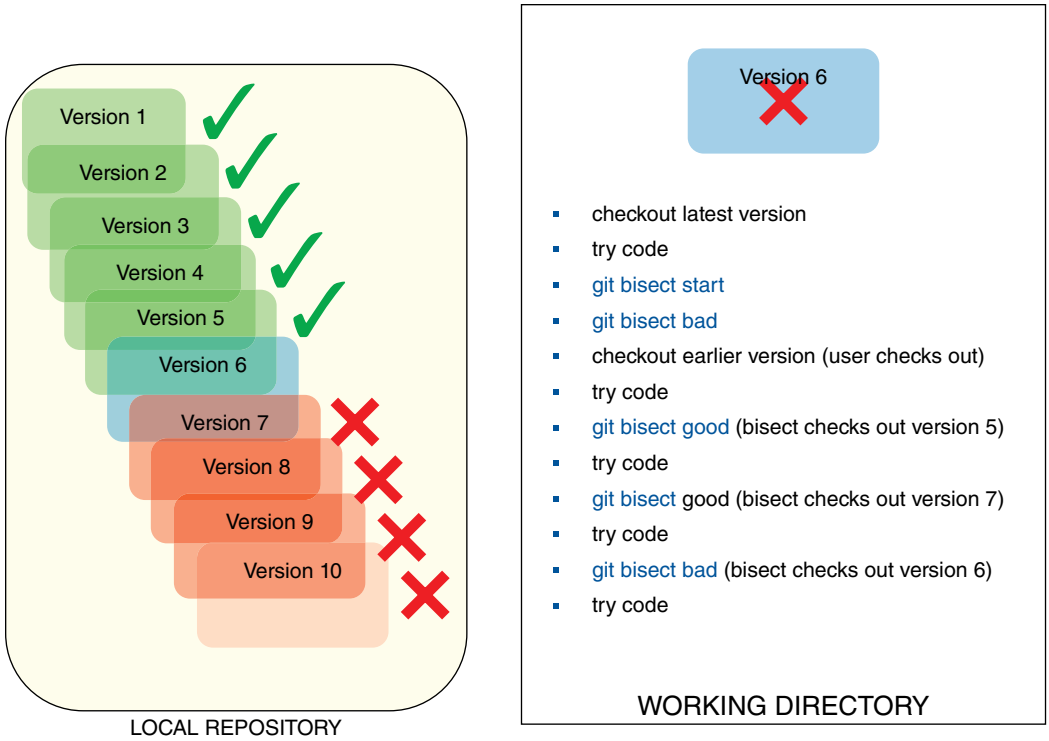


FIGURE 11-13: Narrowing in on the first bad commit

From this examination of the last commit, Git can now determine that this was the first bad commit, and it tells you that.

You have additional interfaces to see the steps and results that are associated with this bisect operation. Assuming that gitk is installed and accessible to your path, you can run *git bisect visualize* and be presented with a view of the bisect operation based on how the pointers have been left in the repository (see Figure 11-15).

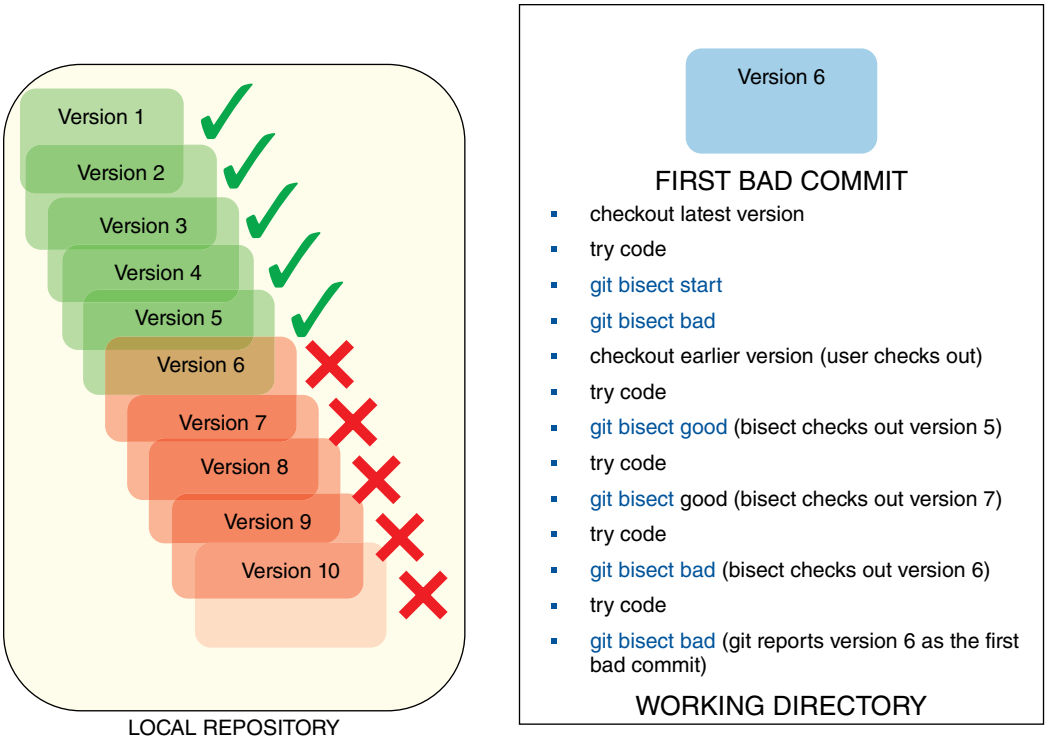


FIGURE 11-14: The first bad commit is found

You can also generate a log of the operations by using *git bisect log*.

Finally, there are times when you may want to leverage automation to help you zero in on the first bad commit. If you have a script that you can run against the set of things in your working directory to indicate success or failure, you can use that script to automate the process. The caveat is that the script must return 0 for a good revision and any code from 1 to 127 for a bad revision, except for 125, which indicates that this commit cannot be tested.

To invoke this kind of automated script, you use the *bisect run* subcommand. A quick example should help to illustrate this. Suppose that you have a simple script that does a test by adding two numbers together. However, you know that somewhere in the various revisions of this script, a bug was introduced that resulted in it subtracting the numbers instead of adding them. By feeding the script to bisect's run subcommand with arguments that should result in a 0, you can find where Git first observes the problem. (If the calculations are correct, Git returns a 0, which indicates to bisect that a commit is okay.) Note that this is not necessarily the same as the first commit where the problem was introduced, but when you have a large set of commits to examine, this process can significantly reduce the search time.

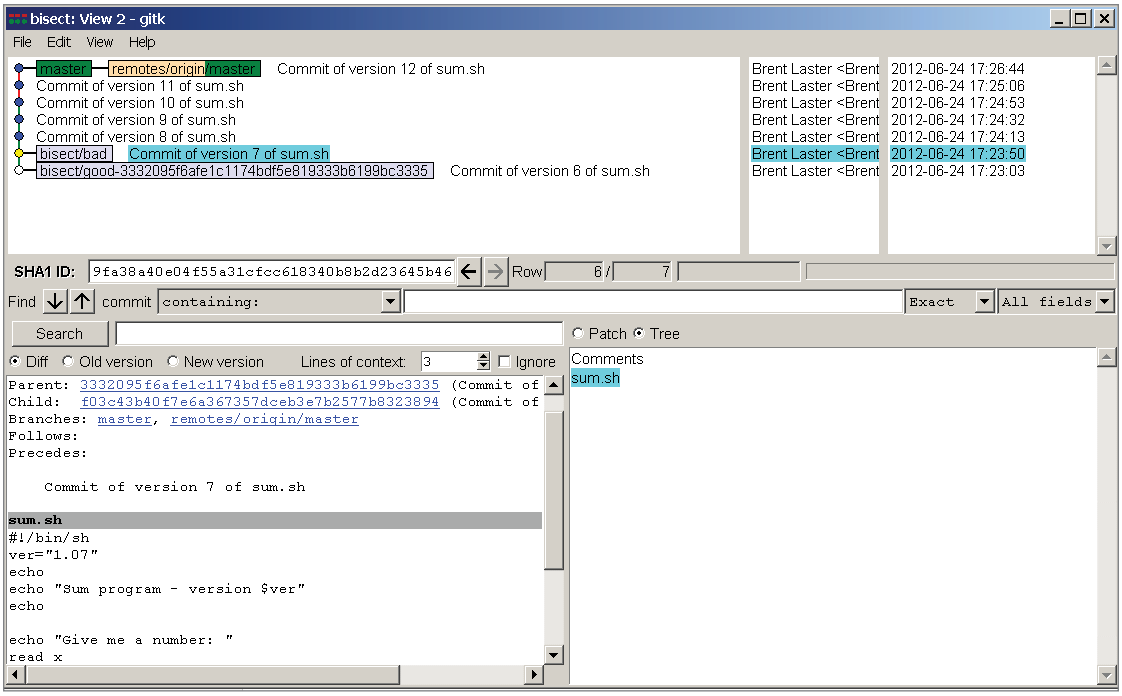


FIGURE 11-15: gitk view of a bisect

The bisect command also has several other subcommands that are detailed in its help page, including ways to tell it to skip specific commits, and more variations on starting ranges.

```
$ git bisect log
git bisect start
# bad: [ada1a94bfcbb6ba448ff38265517eb64bc55a93d] Commit of version 12 of sum.sh
git bisect bad ada1a94bfcbb6ba448ff38265517eb64bc55a93d
# good: [aea87ad663b225b8723e9db828883d0e102f3487] Commit of version 1 of sum.sh
git bisect good aea87ad663b225b8723e9db828883d0e102f3487
# good: [3332095f6afe1c1174bdf5e819333b6199bc3335] Commit of version 6 of sum.sh
git bisect good 3332095f6afe1c1174bdf5e819333b6199bc3335
# bad: [0cd39f3c4927aa0316147548ab5480e76db361e1] Commit of version 9 of sum.sh
git bisect bad 0cd39f3c4927aa0316147548ab5480e76db361e1
# bad: [f03c43b40f7e6a367357dceb3e7b2577b8323894] Commit of version 8 of sum.sh
git bisect bad f03c43b40f7e6a367357dceb3e7b2577b8323894
# bad: [9fa38a40e04f55a31cfcc618340b8b2d23645b46] Commit of version 7 of sum.sh
git bisect bad 9fa38a40e04f55a31cfcc618340b8b2d23645b46
# first bad commit: [9fa38a40e04f55a31cfcc618340b8b2d23645b46] Commit of version 7
of sum.sh
```

```
$ git bisect run ./sum.sh -2 2
running ./sum.sh -2 2

Sum program - version 2.06

-2 + 2 = 0
Bisecting: 2 revisions left to test after this (roughly 2 steps)
[078e5c123e86bfde3396eb1a8adc823eb940e6b3] new version 2.09
running ./sum.sh -2 2

Sum program - version 2.09

-2 + 2 = -4
bisect run failed:
exit code 252 from './sum.sh -2 2' is < 0 or >= 128
```

rerere

Although its name may look like a typo, *rerere* is an actual command in Git. It is short for *reuse recorded resolution*. There are different schools of thought on how to actually pronounce it; some say *ree-ree-ree* and others *rear-er* or *re-were*. My personal choice is *re-3,* which refers to *re* being repeated three times.

With rerere, when you first encounter a conflict from one of the merge-style operations, you resolve the conflict as appropriate, but you tell Git to learn and remember how you resolved that conflict. Afterward, if the same conflict occurs again, and rerere is enabled, Git will resolve the conflict automatically in the same way you did.

In order to use this functionality, you first have to enable it. You do this by setting a special configuration value.

```
$ git config --global rerere.enabled 1
```

This command is a little different from the other Git commands that you've been working with. The rerere functionality, once enabled, runs automatically. However, there are options that allow you to modify the state of the remembered information and to understand what rerere knows. The syntax is as follows:

```
git rerere [clear|forget <pathspec>|diff|remaining|status|gc]
```

I'll talk about some of these options as I go along.

So, what's the practical application of the rerere command? After all, if you resolve a conflict, aren't you done? Consider that you have a topic or feature branch where you are iterating over time, but as you make progress, you want to merge back the latest changes periodically into another branch. If you repeatedly encounter the same merge conflicts each time you merge, then rerere can simplify things significantly.

In a related manner, if you have a long-lived branch that you are working with, from time to time, you may want to merge in the latest changes from the production branch with rerere enabled and then reset back to before the merge. This has the effect of teaching Git how to deal with any conflicts from the history of the production branch when it's finally time to fully merge the long-lived

branch into the production one. This results in fewer delays and less complication because Git will have learned through the temporary merges how to resolve conflicts along the way.

Let's look at an example from a demo project I use in some of my workshops. This project is called *greetings* and consists of a single java file that prints out some text. It also has two branches, master and topic1, with the file changed on each branch, so you encounter a conflict when you merge.

You've enabled rerere using the previous command. Now let's try to merge topic1 into master. (Your current branch is already master.)

```
$ git merge topic1
Auto-merging helloWorkshop.java
CONFLICT (content): Merge conflict in helloWorkshop.java
Recorded preimage for 'helloWorkshop.java'
Automatic merge failed; fix conflicts and then commit the result.
```

This looks similar to what you've seen before when dealing with conflicts, except for the line about "Recorded preimage …". Because you have rerere enabled and there's a conflict, Git records what the file looks like before you resolve the conflict.

Let's take a look at the contents of the file with the conflicts, as marked by Git:

```
$ cat helloWorkshop.java
/* Hello Workshop Java Program */
/* 9-12-12 */
class helloWorkshop  {
    public static void main(String[] args) {
<<<<<<< HEAD
        System.out.println("Greetings Workshop!");
=======
        System.out.println("Greetings People!");
>>>>>>> topic1
        System.out.println("This is an example file");
        System.out.println("for use in workshop exercises.");
        System.out.println("This is on the topic1 branch");
        System.out.println("This file is written in Java.");
        System.out.println("Isn't this exciting?");
<<<<<<< HEAD
        System.out.println("Maybe?");
=======
        System.out.println("Really?");
>>>>>>> topic1
        System.out.println("Okay then.");
        System.out.println("Goodbye Workshop!");
    }
}
```

In this case, it's easy to see which file is involved. However, what if there were multiple files scattered across directories with conflicts? The rerere command includes a status command, similar to the git status command, to tell you which files have conflicts that rerere will monitor and record the resolution for—just the one file in this instance.

```
$ git rerere status
helloWorkshop.java
```

And, as you progress through resolving differences, you can use the diff option to rerere to see the diffs for the current state of the conflict resolution.

```
$ git rerere diff
--- a/helloWorkshop.java
+++ b/helloWorkshop.java
@@ -2,21 +2,21 @@
 /* 9-12-12 */
 class helloWorkshop  {
     public static void main(String[] args) {
-<<<<<<<
-        System.out.println("Greetings People!");
-=======
+<<<<<<< HEAD
         System.out.println("Greetings Workshop!");
->>>>>>>
+=======
+        System.out.println("Greetings People!");
+>>>>>>> topic1
         System.out.println("This is an example file");
         System.out.println("for use in workshop exercises.");
         System.out.println("This is on the topic1 branch");
         System.out.println("This file is written in Java.");
         System.out.println("Isn't this exciting?");
-<<<<<<<
+<<<<<<< HEAD
         System.out.println("Maybe?");
-=======
+=======
         System.out.println("Really?");
->>>>>>>
+>>>>>>> topic1
         System.out.println("Okay then.");
         System.out.println("Goodbye Workshop!");
     }
```

Let's assume that you resolve the conflicts manually this first time, and you end up with this version:

```
$ cat helloWorkshop.java
/* Hello Workshop Java Program */
/* 9-12-12 */
class helloWorkshop  {
    public static void main(String[] args) {
        System.out.println("Greetings Workshop!");
        System.out.println("This is an example file");
        System.out.println("for use in workshop exercises.");
        System.out.println("This is on the topic1 branch");
        System.out.println("This file is written in Java.");
        System.out.println("Isn't this exciting?");
        System.out.println("Really?");
        System.out.println("Okay then.");
        System.out.println("Goodbye Workshop!");
    }
}
```

Now you can stage and commit the change.

```
$ git add .
$ git commit -m "fixed"
```

```
Recorded resolution for 'helloWorkshop.java'.
[master 8d8bbcb] fixed
```

Notice the additional message you get from Git during the commit about "Recorded resolution…". Git is telling you that because the rerere function is enabled, and you have resolved the conflict resolution (as indicated by the commit), it has recorded how the file was changed to resolve the conflict.

NOTE *To implement the rerere functionality, Git creates a datastore in the .git directory named rr-cache. (In fact, another way to activate the rerere functionality instead of through the git config command is to just create the .git/rr-cache area.) In this area, Git stores the pre-image and post image of each file rerere knows about, organized by SHA1. The pre-image is the copy of the file with the conflicts. The post image is the copy of the file after the conflicts have been resolved.*

The following commands and listings show examples of the contents and layout of an rr-cache:

```
$ ls .git/rr-cache
e6b62c2cbff74f18e090ced201888bf2e8b44300/

$ ls .git/rr-cache/e6b*
postimage  preimage

$ cat .git/rr-cache/e6b*/preimage

/* Hello Workshop Java Program */
/* 9-12-12 */
class helloWorkshop  {
    public static void main(String[] args) {
<<<<<<<
        System.out.println("Greetings People!");
=======
        System.out.println("Greetings Workshop!");
>>>>>>>
        System.out.println("This is an example file");
        System.out.println("for use in workshop exercises.");
        System.out.println("This is on the topic1 branch");
        System.out.println("This file is written in Java.");
        System.out.println("Isn't this exciting?");
<<<<<<<
        System.out.println("Maybe?");
=======
        System.out.println("Really?");
>>>>>>>
        System.out.println("Okay then.");
        System.out.println("Goodbye Workshop!");

    }
}

$ cat .git/rr-cache/e6b*/postimage
/* Hello Workshop Java Program */
/* 9-12-12 */
```

```
class helloWorkshop  {

public static void main(String[] args) {
    System.out.println("Greetings People!");
    System.out.println("This is an example file");
    System.out.println("for use in workshop exercises.");
    System.out.println("This is on the topic1 branch");
    System.out.println("This file is written in Java.");
    System.out.println("Isn't this exciting?");
    System.out.println("Really?");
    System.out.println("Okay then.");
    System.out.println("Goodbye Workshop!");
}
}
```

As an example of what rerere can do for you after it has recorded a resolution, let's reset back to before the merge and do it again. You'll use the ORIG_HEAD option to go back, as discussed in the main part of this chapter.

```
$ git reset --hard ORIG_HEAD
HEAD is now at da7a1e2 update for master
```

Doing the merge again, you get the following output:

```
$ git merge topic1
Auto-merging helloWorkshop.java
CONFLICT (content): Merge conflict in helloWorkshop.java
Resolved 'helloWorkshop.java' using previous resolution.
Automatic merge failed; fix conflicts and then commit the result.
```

Notice the line, "Resolved … using previous resolution." Because rerere is enabled, and you had shown it before how to resolve this particular conflict, Git simply resolves it, in the same way you did manually before. In fact, if you take a look at the contents, you see that they look just like the version you ended up with after the earlier manual editing.

```
$ cat helloWorkshop.java
/* Hello Workshop Java Program */
/* 9-12-12 */
class helloWorkshop  {
    public static void main(String[] args) {
        System.out.println("Greetings Workshop!");
        System.out.println("This is an example file");
        System.out.println("for use in workshop exercises.");
        System.out.println("This is on the topic1 branch");
        System.out.println("This file is written in Java.");
        System.out.println("Isn't this exciting?");
        System.out.println("Really?");
        System.out.println("Okay then.");
        System.out.println("Goodbye Workshop!");
    }
}
```

And, if you check the status and diffs that rerere is tracking, you see that there are no additional conflicts it knows about.

```
$ git rerere status
$ git rerere diff
```

You may be wondering why, if Git was able to resolve the conflict, it still reported that the automatic merge failed and that you need to fix conflicts. This is just a general message that the merge-style operation provides. There may be other conflicts that rerere hasn't been taught how to resolve. You may also want to review the affected files and make other changes. Or, you may not want the automatic resolution for a particular case after all.

In the case of automatic resolution, you can use one of the other options with rerere to undo the automatic resolving functionality. The forget option with a path tells Git to forget the resolution for that particular file.

```
$ git rerere forget helloWorkshop.java
```

The forget option also allows you a kind of redo if the recorded resolution isn't what you wanted. If you realize that a resolution you did was incorrect or not what you wanted, you can follow this sequence:

1. Tell Git to forget the resolution—`git rerere forget <file>`
2. Create the conflict version again—`git checkout --merge <file>`
3. Resolve the conflicts as desired for `<file>`
4. Run rerere again to record the new resolution—`git rerere`

NOTE *As its name implies, the merge option on checkout tells Git to attempt a merge when it does a checkout. This is primarily intended to allow for switching branches when you are doing a checkout and there are unmerged changes in your current branch. Normally, Git does not allow this and you must get to a clean state by committing, merging, or stashing (discussed in Chapter 9) those changes.*

However, the merge option also has a useful secondary function: reproducing a file with conflicts marked in it if needed. So, if you have resolved conflicts locally, but then change your mind, you can get back to the version of the file with conflicts by doing another checkout on top of the current one and adding the --merge option.

A related checkout option is --conflict. This option is like the merge option but allows you to specify a style. *A style in this case defines how conflicts are marked and displayed.*

The two options are --conflict=merge (the same as --merge) and --conflict=diff3 (which shows the two versions and the common ancestor version, if different).

Let's look at a quick example using the file you previously used with the rerere workflow, helloWorkshop.java:

```
$ cat helloWorkshop.java
/* Hello Workshop Java Program */
```

```
/* 9-12-12 */
class helloWorkshop  {
    public static void main(String[] args) {
<<<<<<< HEAD
        System.out.println("Greetings Workshop!");
=======
        System.out.println("Greetings People!");
>>>>>>> topic1
        System.out.println("This is an example file");
        System.out.println("for use in workshop exercises.");
        System.out.println("This is on the topic1 branch");
        System.out.println("This file is written in Java.");
        System.out.println("Isn't this exciting?");
<<<<<<< HEAD
        System.out.println("Maybe?");
=======
        System.out.println("Really?");
>>>>>>> topic1
        System.out.println("Okay then.");
        System.out.println("Goodbye Workshop!");
    }
}
```

If you now do the checkout again with the --merge option or the --conflict=merge option, you get a similar representation:

```
$ git checkout --conflict=merge helloWorkshop.java
$ cat helloWorkshop.java
/* Hello Workshop Java Program */
/* 9-12-12 */
class helloWorkshop  {
    public static void main(String[] args) {
<<<<<<< ours
        System.out.println("Greetings Workshop!");
=======
        System.out.println("Greetings People!");
>>>>>>> theirs
        System.out.println("This is an example file");
        System.out.println("for use in workshop exercises.");
        System.out.println("This is on the topic1 branch");
        System.out.println("This file is written in Java.");
        System.out.println("Isn't this exciting?");
<<<<<<< ours
        System.out.println("Maybe?");
=======
        System.out.println("Really?");
>>>>>>> theirs
        System.out.println("Okay then.");
        System.out.println("Goodbye Workshop!");
    }
}
```

Notice that the different branches are marked with theirs and ours, which correspond to the merge strategy options that you could use to overwrite from one branch or the other.

A git checkout with the --conflict=diff3 option produces a similar file but also adds in the base *version—the version before changes have occurred on the two branches (assuming the file has been changed on both branches). Think of this as the common ancestor I talked about earlier when Git does a three-way merge—thus the 3 in diff3.*

```
$ git checkout --conflict=diff3 helloWorkshop.java

$ cat helloWorkshop.java
/* Hello Workshop Java Program */
/* 9-12-12 */
class helloWorkshop  {
    public static void main(String[] args) {
<<<<<<< ours
        System.out.println("Greetings Workshop!");
||||||| base
        System.out.println("Hello Workshop!");
=======
        System.out.println("Greetings People!");
>>>>>>> theirs
        System.out.println("This is an example file");
        System.out.println("for use in workshop exercises.");
        System.out.println("This is on the topic1 branch");
        System.out.println("This file is written in Java.");
        System.out.println("Isn't this exciting?");
<<<<<<< ours
        System.out.println("Maybe?");
||||||| base
        System.out.println("No?");
=======
        System.out.println("Really?");
>>>>>>> theirs
        System.out.println("Okay then.");
        System.out.println("Goodbye Workshop!");
    }
}
```

The following code is an example of the commands that are involved in updating a resolution that you previously taught rerere:

```
$ git rerere forget helloWorkshop.java
$ git checkout --merge helloWorkshop.java
<edit and resolve conflicts as desired>
$ git rerere
```

With these commands, you told Git to forget the previous resolution, re-created the file with conflicts, resolved things the way you wanted, and then told Git to record the new resolution.

There is one other option you can enable for rerere that is a useful shortcut: automatically staging files that rerere was able to resolve successfully. You do this by setting another configuration value.

```
$ git config --global rerere.autoupdate true
```

If you go back and redo the same merge later such that the rerere command can automatically resolve the conflicts, you will see something like this:

```
$ git merge topic1
Auto-merging helloWorkshop.java
CONFLICT (content): Merge conflict in helloWorkshop.java
Staged 'helloWorkshop.java' using previous resolution.
Automatic merge failed; fix conflicts and then commit the result.
```

Note the line that says "Staged 'helloWorkshop.java' using previous resolution." This means that because Git was able to resolve the conflicts with rerere, it staged the resolved file for you. And, in fact, if you check the normal Git status, you see your staged file.

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
All conflicts fixed but you are still merging.

Changes to be committed:

        modified:   helloWorkshop.java
```

SUMMARY

In this chapter, you explored a set of less-common but useful commands for working with content in Git. I spent some time talking about the git stash command, which allows you to move uncommitted changes from the working tree and staging area into an element in a separate queue. It then resets your local environment back to the state of the last commit. Then, if you need to get those elements back out of the queue, you can apply or pop them back into your local environment.

You also learned about the mv and rm commands for renaming and deleting content. You looked at how these commands make local changes in the working directory, and stage the change. As with file or other content updates, you have to commit to make the change happen in the repository.

You looked at the grep command in Git (which is much like the OS system grep command) that allows you to search through content for strings or regular expressions. I discussed how to search through the Git history log and how to use the -S or pickaxe option for extended searching.

I covered some examples of using commands like bundle and archive to create external packages of Git content to share. I also explained how to share patches through e-mail directly with Git.

Finally, in the main part of the chapter, I covered the notes functionality in Git that allows you to add notes to commits that are already in the repository.

In the advanced topics section, I covered three more complicated commands that provide a lot of power for users. The filter-branch command allows you to re-process content in repositories to do

things like split out parts of repositories into their own separate repositories, or change attributes associated with commits.

The bisect command uses a form of binary searching across commits in a repository to help users narrow in on where a particular change was first introduced.

The rerere command allows you to teach Git how to resolve merge situations so it can remember the resolution if you encounter the same situation again, and automatically resolve it for you in the way you want.

In the next chapter, I cover the remote side of the Git environment and talk about remote repositories, remote branches, and interactions with them to complete your overall Git workflow.

About Connected Lab 7: Deleting, Renaming, and Stashing

This lab guides you through some simple examples of using several of the basic commands covered in this chapter. After each of the change steps, you'll look at the status of the files in your working directory to understand what's changed. This will help you to get a feel for these kind of operations as part of an overall workflow using Git.

Connected Lab 7

Deleting, Renaming, and Stashing

In this lab, you'll work through some examples of using the rm and mv commands and stashing uncommitted changes.

PREREQUISITES

This lab assumes that you have done Connected Lab 6: Practicing with Merging. You should start out in the same directory as that lab.

STEPS

1. Starting out in the same directory as Connected Lab 6, create a new file, stage it, and commit it.

   ```
   $ echo "another one" > file6.c
   ```

   ```
   $ git add .
   ```

   ```
   $ git commit -m "yet another lab file"
   ```

2. You now decide to remove the file. Use the rm command to do that.

   ```
   $ git rm file6.c
   ```

3. Check the status to determine whether the file is staged for removal.

   ```
   $ git status
   ```

4. Run the ls command to find out whether the local file is still there.

   ```
   $ ls
   ```

5. You now change your mind, and you want the file back. Use the reset command to do that.

```
$ git reset --hard HEAD
```

6. Check the status.

```
$ git status
```

7. To find out whether the file is back locally, run the ls command.

```
$ ls
```

8. Delete the file anyway.

```
$ git rm file6.c
```

9. Check the status; the file should be staged for deletion.

```
$ git status -sb
```

10. You commit to the deletion this time.

```
$ git commit -m "<comment>"
```

11. (Optional) Check the status and run the ls command to make sure the file is really gone.

```
$ git status
$ ls
```

12. (Optional) Take a look at the changes in gitk. (Start it using the command "gitk &" if you don't have gitk running, or press F5 to refresh if you do have it open.)

13. You decide to make a couple of other changes. Start by creating a new file.

```
$ echo "one more" > file7.c
```

14. Stage the file.

```
$ git add file7.c
```

15. Check the status.

```
$ git status
```

16. You need to quit working with this file temporarily and fix something else. To do this, save off the current state with the stash command.

```
$ git stash
```

17. Check the status and local directory for the new file you were working with.

```
$ git status
$ ls
```

18. Take a look at what's in the stash.

```
$ git stash list
```

19. Make a change to an existing file, and then stage and commit it.

```
$ echo update >> file5.c
$ git commit -am "update file"
```

20. You're done with your change, so restore your previous state.

```
$ git stash pop
```

21. Check the status and the local directory to make sure everything is the way it was.

```
$ git status
$ ls
```

12

Understanding Remotes—Branches and Operations

WHAT'S IN THIS CHAPTER?

- Understanding the different meanings of *remote* in Git
- Using networking protocols with Git
- Adding and managing remote references
- Understanding remote tracking branches
- Learning more about the clone command
- Establishing connections between local and remote branches
- Pushing, fetching, and pulling changes

Up until now, the examples that I've looked at, and the majority of things that I've talked about, have involved working with Git in the local environment. I previously defined the local environment as the three Git-related areas that reside on your local system—the working directory, the staging area, and the local repository—along with the supporting pieces, such as configuration, that work with them.

It is a departure from most other texts on Git to wait this long before diving more deeply into the remote side of Git and the remote repository. However, if you have been reading this book in the order it was written to learn about Git, you will have developed a firm foundation to help you understand the interactions with the remote side.

By now, you should be comfortable with how Git works in the local environment, and also understand how it manages those three levels that make up the local environment. And, certainly, you should be well acquainted with the warnings against modifying content that has already been put into the remote environment (the remote repository).

With that foundation, let's take a closer look at what I mean by *remotes*.

REMOTES

Whenever you talk about *remotes* in Git, this word can have several meanings. Most commonly, it refers to a remote repository. In its basic form, a remote repository is just a Git repository with a protocol for access and server-side hooks. As I discuss in Chapter 3, you can think of the remote repository as the public or server-side repository. In most other source management systems, this would be the only repository you have.

> **NOTE** *There is one more characteristic of remote repositories: they are* bare. *Bare in this context means that they do not have a checked-out set of content associated with them. They are not intended to have branches checked out from them. They exist for the purpose of tracking and synching content in a repository only.*
>
> *The clone command in Git can take a* --bare *option to get a copy of the repository as it would be on the remote side. This can be useful, for example, to create a copy of the repository that is suited for migrating to another remote location.*

The remote repository is also the *collection point* for code from multiple users. If you are working with multiple people, each working in their own local environment, they may all be pushing code to share with others to a common remote repository (see Figure 12-1).

> **NOTE** *Nothing prevents you from having a remote repository instantiated on your local system. That may seem confusing at first, but consider that you could similarly have a private CVS or subversion server running on your local system to commit changes to. That system may just be for you to connect to, or you may open it up for others. In this case, remote is usually thought of as the shared, public repository for code, regardless of where it is actually hosted.*

Another common use case of the term *remote* in Git is a local *reference* to a remote repository. Remote repositories typically have Internet-accessible addresses composed of URLs formatted for a particular network protocol, such as SSH, HTTP, or HTTPS. (I discuss these protocols in more detail in the following section.) Those addresses can be long, which can make them difficult to remember as well as difficult to type in with commands that need to access them. So, once a connection has been established, Git allows the user to use simple, one-word reference names to point to the URL. The default reference name in Git is *origin*.

As an example, I might have a Git repository stored on

```
https://corporateGitServer/BrentLaster/demoprojects/project1.git
```

That's a long URL to remember and type, and so it's difficult to get right every time I want to use some command to access the remote at that location. So, when I clone a copy of the project down to my local machine to work with, Git establishes the remote reference name as *origin*. This means that instead of having to type

```
$ git push https://corporateGitServer/BrentLaster/demoprojects/project1.git ...
```

I can simply type

```
$ git push origin ...
```

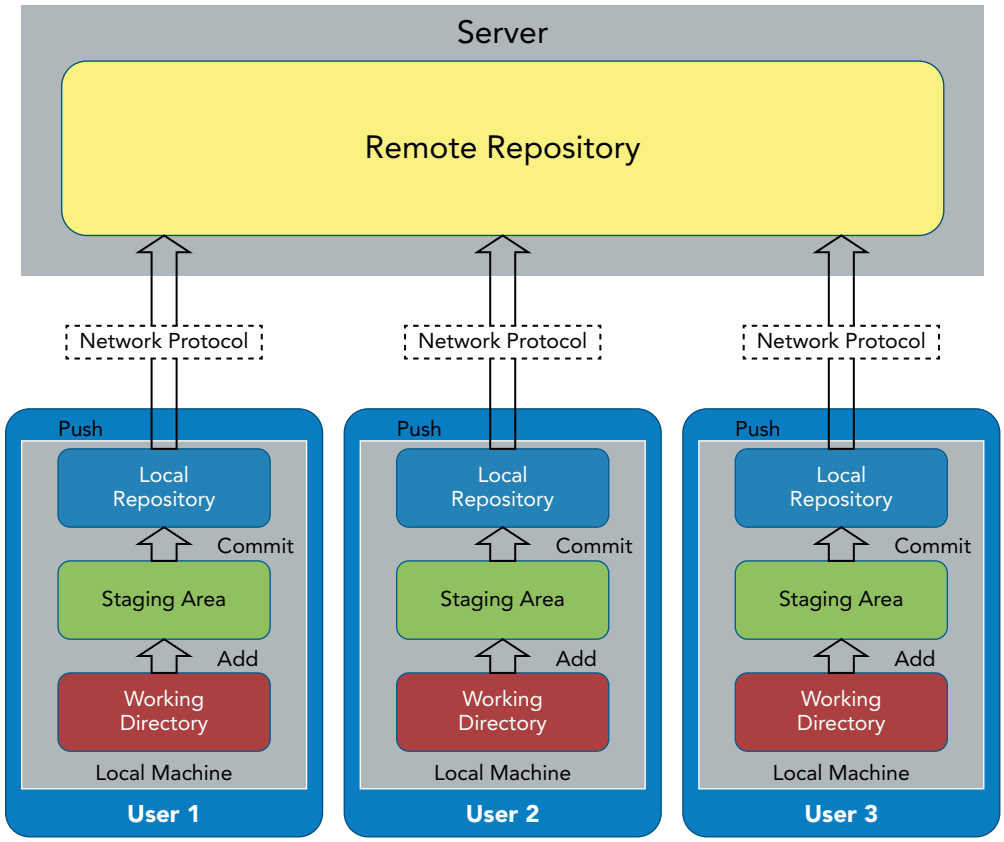


FIGURE 12-1: Arrangement of local versus remote environments

These remote reference names are one-way references—names that point to a location. Removing the name does not affect the remote repository; the names are just aliases for the repositories.

Remote Access Protocols

I'll now take a moment and talk about the various networking protocols that you can use to communicate with a Git remote. There are four available protocols: *Local, Git, SSH*, and *HTTP*.

Local

The Local protocol is essentially just filesystem access over something like a Network File System (NFS) mount or a shared drive. It can be convenient to share content between multiple users using an easily accessible location. However, this protocol is not authenticated or protected (other than as defined by your file system access). So, using this protocol involves the same risk as relying on any open access in a shared location. Also, it is only convenient as long as you are able to connect (and stay connected) to the shared resource.

An example of cloning down something using the Local protocol might be

```
$ git clone /var/git/repos/myproj.git
```

There is no special setup for using this protocol, other than providing the shared access.

Git

Git comes with a special daemon program that you can use to provide access to a repository over a dedicated port (9418). This is the fastest protocol, but it comes with some big drawbacks: no authentication and an *all-or-none* access model (if anyone can access it, everyone can access it). Setting up access through the port can also be difficult.

An example of cloning down something using this protocol might be

```
$ git clone git://repos/myproj.git
```

Setup for using this protocol generally involves installing a daemon, starting it up, and adding a special *marker* file to each project so that Git knows that it's okay for the daemon to access it.

To understand more about the use of the daemon program, see the help page for git-daemon.

```
$ git daemon --help
```

SSH

Secure Shell (SSH) is a more commonly used protocol. It is also one that most users and network administrators are familiar with; it is also inexpensive to set up and use.

If you're not familiar with SSH, it operates on the idea of authenticated access using private and public *keys*. The public key goes on the resource you need access to (the Git remote in this case),

and the private key goes into a special ~/.ssh subdirectory on your local machine. As long as the private key on the user's system corresponds to a public key on the desired resource, the user can connect and transfer information without having to log in each time. Authentication is handled using the keys.

There are two forms of paths that you can use with SSH:

```
$ git clone ssh://<username>@mygitserver.mycompany.com/myproject.git
```

or

```
$ git clone <username>@mygitserver.mycompany.com:myproject.git
```

Figure 12-2 illustrates the difference between the traditional login methods of access (top) and SSH access with keys (bottom).

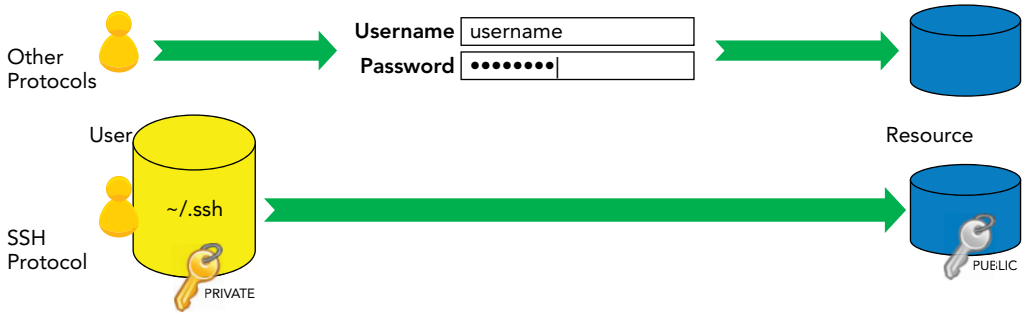


FIGURE 12-2: Login access (top) versus SSH access (bottom)

HTTP

There are two types of HTTP access that you can use with Git: dumb HTTP and smart HTTP. The dumb HTTP mode, which is read-only, is simply handled and served like any other items under an HTTP server. To set it up, a Git repository is put under an HTTP document root, and a special Git hook (post-update) is set up.

The smart HTTP mode allows for read-and-write access using authentication or anonymous access. It is more similar to SSH in terms of operation but doesn't require the keys; instead, it can use standard username and password authentication. A common URL can be used for viewing, cloning, and pushing changes (assuming you have access).

Setup of the smart HTTP mode involves setting up the web configuration and permissions, in addition to having requests handled by a CGI script named *git-http-backend*, which is included with Git. Here is an example of cloning with this protocol:

```
$ git clone https://mycompany.com/repos/myproject.git
```

TIP *Git provides built-in ways to help with credentials. There are also external programs to help with them.*

Credentials *refer to values supplied for authenticating access to a resource. Typically, this means usernames and passwords. Helping with these values usually equates to remembering or storing them for some period of time so that the user does not have to enter them (authenticate) as frequently.*

The built-in mechanism can store the username in the configuration to avoid having to enter it frequently. It can also be told to cache values for some period of time.

Your Git instance may already have credential helpers involved. You can check by looking for credential- *in the output of the extended help command.*

```
$ git help -a | grep "credential-"
```

If you have credential helpers available, you can find out more about them by using the specific help command.

```
$ git help credential-<name>
```

Then, if you decide to use this helper application, you can set the configuration value credential.helper to specify that particular application.

```
$ git config --global credential.helper <name>
```

As I have noted, external credential helper programs are also available. Consult their specific documentation for how to use them.

Now that you understand the basic concepts of remotes, you can look at the git remote command.

The Remote Command

The git remote command allows you to manage your connections and interactions with remote repositories. The syntax is as follows:

```
git remote [-v | --verbose]
git remote add [-t <branch>] [-m <master>] [-f] [--[no-]tags]
[--mirror=<fetch|push>] <name> <url>
git remote rename <old> <new>
git remote remove <name>
git remote set-head <name> (-a | --auto | -d | --delete | <branch>)
git remote set-branches [--add] <name> <branch>...
git remote get-url [--push] [--all] <name>
git remote set-url [--push] <name> <newurl> [<oldurl>]
git remote set-url --add [--push] <name> <newurl>
```

```
git remote set-url --delete [--push] <name> <url>
git remote [-v | --verbose] show [-n] <name>...
git remote prune [-n | --dry-run] <name>...
git remote [-v | --verbose] update [-p | --prune] [(<group> | <remote>)...]
```

As you can see, there are a lot of options here, but you will probably only use a few of them. Many of these options relate to the idea of a *remote reference* that I talked about earlier in this chapter; this is a short name that refers to the longer protocol string (https://..., git://..., ssh://...) that is the actual location of the remote repository.

Let's look at a couple of the common use cases for the git remote command. As I mentioned, when you clone down a remote repository, Git automatically sets up a remote (reference) named *origin* that maps to whatever location you cloned from. You can see what a remote maps to at any point by using the -v or --verbose option of the command. Here is an example:

```
$ git clone https://github.com/brentlaster/gradle-greetings

$ cd gradle-greetings

$ git remote -v
origin  https://github.com/brentlaster/gradle-greetings (fetch)
origin  https://github.com/brentlaster/gradle-greetings (push)
```

Now, for any of the commands that I need to use to interact with the remote repository at http://github.com/brentlaster/gradle-greetings, I can just use *origin* instead of typing out that longer path.

I can also add, remove, and rename remote references. Again, these are just references or *aliases*. Modifying these references does not affect the remote repository.

```
git remote add version2 https://github.com/brentlaster/gradle-greetings-sets

$ git remote -v
origin  https://github.com/brentlaster/gradle-greetings (fetch)
origin  https://github.com/brentlaster/gradle-greetings (push)
version2        https://github.com/brentlaster/gradle-greetings-sets (fetch)
version2        https://github.com/brentlaster/gradle-greetings-sets (push)

$ git remote rename version2 origin2

$ git remote -v
origin  https://github.com/brentlaster/gradle-greetings (fetch)
origin  https://github.com/brentlaster/gradle-greetings (push)
origin2 https://github.com/brentlaster/gradle-greetings-sets (fetch)
origin2 https://github.com/brentlaster/gradle-greetings-sets (push)

$ git remote rm origin2

$ git remote -v
origin  https://github.com/brentlaster/gradle-greetings (fetch)
origin  https://github.com/brentlaster/gradle-greetings (push)
```

NOTE *Based on that last example, you may be wondering why you would ever have multiple remotes associated with a local environment. There are a few common use cases:*

- *You are working on code targeted for one remote repository, but you want to be able to merge in or use code from another remote repository.*
- *You are working on making an updated or customized version of an existing project and pushing changes out to your updated or customized remote. However, you want to maintain a connection to the existing project in case pieces of it are updated that you may want or need to incorporate in your version.*
- *You are working on very similar changes suitable for multiple remotes.*

Notice that there are also options to get and set the URL associated with the remote. This can be useful if you cloned with one protocol but want to use a different one now.

Now that you understand the basic idea of remotes, let's look at how Git interacts with them and how you keep track locally of where things are on the remote side.

How Git Interacts with the Remote Environment

Remote repositories in Git are dumb. This doesn't mean they're a bad idea, but rather that they don't have to do a lot of processing. So, you could say that the model around them is smart.

Git does not maintain a constant connection from the local repository to the remote repository. (If it did and it required this, you wouldn't be able to do disconnected development.) Rather, Git checks in with the remote repository to get updated status information and contents. This checking in occurs whenever a Git command that requires interaction with the remote repository is initiated in the local environment—essentially, when doing a fetch, pull, or push operation.

For the necessary duration of those operations, Git in the local environment establishes the connection to the remote repository, gathers information about where branches of interest are in the remote repository, and updates or downloads content as appropriate.

So, Git can get by with just temporary connections to the remote environment. This is similar to how most traditional source management systems work. A connection to the server is only established and used when updating content between the server and the local working area.

Remote Tracking Branches

Having the connection between the local environment and the remote environment as a temporary one, only activated when an operation demands, is a useful construct for keeping things simple. It also enables you to limit overhead and bandwidth, and support paradigms like disconnected development.

However, it is not as useful if you are working in your local environment and you want to know how your local changes compare to the versions of changes in the remote repository (status) or you

want to pull content from the remote repository but not merge it in yet. These kinds of operations require some persisted knowledge of the state of the remote repository (between operations that establish physical connections to the remote environment).

Git persists this information about the state of the remote repository by setting up *remote tracking branches* in the local repository. Essentially, the local repository contains state information about the remote repository based on the last time you talked with it (connected to the remote).

These *copies* of the remote branches exist alongside the local branches in the local repository. As such, there has to be a way to distinguish the two branch types because you will usually have local branches with the same name as remote branches. For example, I will have a *master* branch on the remote and a *master* branch in the local repository.

To help distinguish the remote tracking branches from the local branches in the local repository, remote tracking branches have a namespace associated with them—the remote reference name, such as origin, origin2, and so on. For example, if your remote reference to the area you cloned from is called *origin*, then the remote tracking branch for master in the local repository will be named *origin/master* (or more accurately, *remotes/origin/master*). The local master branch will just be *master*. Table 12-1 summarizes the characteristics of the various branches.

TABLE 12-1: Summarizing the Types of Branches in Git

CATEGORY	WHERE IT LIVES	DESCRIPTION	HOW IT IS UPDATED	REFERENCE
Local	Local repository	*Private* branches created by a user in the local environment	By commits, merges, rebases, and so on, in the local environment	<branch name> Example: master
Remote	Remote repository	*Public* branches that track updates from multiple users	By pushes from the users' local environments	Usually <branch name> on <remote reference> Example: master on origin
Remote tracking	Local repository	*Private* branches that *track* the state of the *public* branch on the remote—as it was the last time you connected to the remote	When a Git operation interacts with the remote—such as push, pull, fetch, or clone	<remote reference>/<branch name> or remotes/<remote reference>/<branch name> Example: origin/master or remotes/origin/master

Let's see how this all fits in with the Git model by looking at the clone command.

Git Clone

As I discuss in the early chapters, cloning is how you start working with an existing Git repository. The idea is that you get a copy of the repository from the server (remote side) down to your local environment (a local directory).

You initiate this copy with the git clone command. The syntax is as follows:

```
git clone [--template=<template_directory>]
          [-l] [-s] [--no-hardlinks] [-q] [-n] [--bare] [--mirror]
          [-o <name>] [-b <name>] [-u <upload-pack>] [--reference <repository>]
          [--dissociate] [--separate-git-dir <git dir>]
          [--depth <depth>] [--[no-]single-branch]
          [--recursive | --recurse-submodules] [--] <repository>
          [<directory>]
```

An example clone command is

```
$ git clone ssh://git@gitserver:repository
```

or

```
$ git clone http://gitserver/repository
```

When you do a clone, several things take place, including the following:

- The .git repository is copied from the server (remote side) to the directory locally. This instantiates a new local environment.
- Git creates *remote tracking branches* in the clone that correspond to all of the branches in the remote repository.
- For the currently active HEAD in the remote, Git creates a corresponding local branch.
- For the currently active HEAD, Git checks out a copy of the corresponding local branch.

Figure 12-3 illustrates the *before* and *after* state of a clone.

Notice that you start out with a remote repository containing a few commits and two separate branches: master and testing. From your point of view, these are the *remote branches*. When you do the clone, you get a copy of the remote repository cloned into a local repository. The existing branches in the remote each get their own *remote tracking branches*, namespaced with the name of the remote—that is, origin/master and origin/testing. For the active HEAD, a local branch, *master*, is created in the local repository and the contents are checked out into the working directory.

Remember that creating a branch is just creating a pointer in Git. A benefit of the remote tracking branches is to mark where the branches were located in the remote, so you can reference where they are pointing locally without having to maintain a persistent connection to the remote.

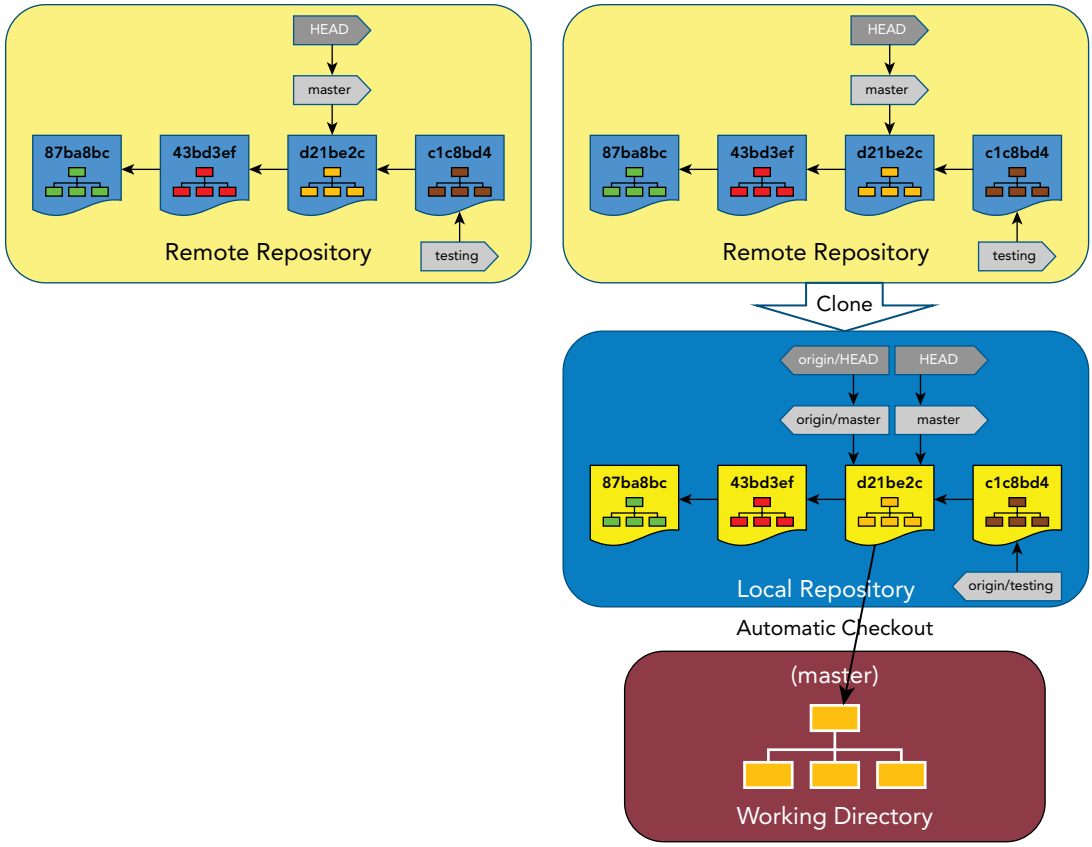


FIGURE 12-3: Start and end of a cloning operation

> **NOTE** *Because I've talked about what the clone command does, I should also mention what it does not do, or rather what it does not clone down. The clone command does not clone down hooks (the programs that run before or after commands) from the remote. The reason for this is that hooks on the remote side are likely not suited for use in local environments. For example, a remote-side hook may have access to mirror content to web servers or to trigger build systems that would not be suitable for a hook to access on the local environment side.*
>
> *(Hooks are covered in more detail in Chapter 15.)*

You can easily see the remote tracking branches after a clone by using the -r option on the *git branch* command.

```
$ git branch -r
  origin/HEAD -> origin/master
  origin/testing
  origin/master
```

Understanding Clone Paths

By default, a clone operation creates a local directory with the name of the repository. If there is a multi-level path, only the last part of the path is the actual repository. You can think of this like downloading a Zip file. The path in front of the Zip file is just the path; the last part is the actual Zip filename. And, when you download the Zip file, it is expanded into whatever structure is contained in the file. Likewise, the last piece of the repository path is the actual repository, and it is expanded into the .git structure and a checked-out version of the current HEAD. Figure 12-4 illustrates this idea.

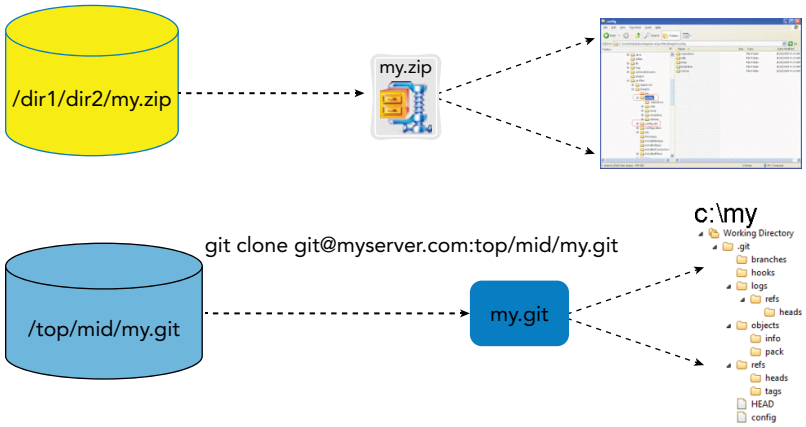


FIGURE 12-4: A way to think about cloning multi-level paths

In most cases, repositories avoid the issue of multiple levels in paths by hyphenating all of the levels together as one name. For example, *top/mid/my.git* becomes *top-mid-my.git*, and the directory that is created when this path is cloned becomes *C:\top-mid-my*.

If you want to clone to a different destination directory (one that's different from the name of the repository), you can simply add that directory as the last argument to the clone command.

```
$ git clone git@myserver.com:top/mid/my.git project1
```

In this case, Git creates the local subdirectory as *project1* and then clones the repository into that subdirectory instead of into *my*.

Clone Options

So far, I've described the basic operation of the clone command. However, there are some other useful options that you should be aware of.

bare

The --bare option tells Git to create a *bare repository*. This is a clone of the repository that is done in a way more suitable for copying or migrating to another remote location. This option has the following characteristics:

- Instead of creating a subdirectory with .git under it, it places the contents that would normally go into the .git directory in the subdirectory itself. So, you end up with repository.git instead of repository/.git.
- The remote branches become local branches in the cloned repository (no remote tracking branches).
- There is no checkout of a branch.

mirror

The --mirror option also tells Git to create a bare repository and implies --bare. However, instead of just copying the remote branches and making them into local branches, it copies all references in the repository, including things like notes or remote-tracking branches if those items are also in the repository. It's really a complete copy of a repository, just done in a Git way.

The mirror option also sets things up so that if you later need to re-copy the repository (thus overwriting it), you can do that with a git remote update command.

branch

The --branch (or -b) option tells Git to make HEAD in the cloned repository point to the branch specified by the option instead of master. This means that branch will also be the one that is checked out.

single-branch

The --single-branch option tells Git to only clone down one branch in the repository. By default, this is whatever HEAD is in the remote, although you can specify another branch using the --branch option.

depth

The depth option creates a shallow clone where the history is truncated to the number of commits specified as an argument to the option. So, --depth=1 would only clone down the latest changes (one commit).

Now that you can clone repositories down and you understand the basic relationships between remote and local branches, let's look at how to view and specify those relationships.

Viewing Information about Remote Branches

After cloning or establishing a connection to a remote branch and synching the content, there are several different ways to see information about remote branches.

First, you can view the list of remote branches with the -r option to git branch. The -r option here refers to *remotes*. Here's an example of how to use it:

```
$ git branch -r
  origin/HEAD -> origin/master
  origin/master
  origin/test
  origin/test2
```

In this case, you have three branches in the remote branch that you are now tracking in the local repository: *master*, *test*, and test2. Notice that this command also shows you where HEAD is pointing.

If you want to see the local branches as well as the remote ones, you can use the -a option (all) to the git branch command.

```
$ git branch -a
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/master
  remotes/origin/test
  remotes/origin/test2
```

Notice that this list now includes your one local branch (master) along with the remote tracking branches.

For any of these options, you can add the -v (verbose) option to see which commit is current on the branch—where the tip of the branch points to. This example shows the short version of the commit's SHA1 and the commit message associated with that commit:

```
git branch -av
* master                539358f update file
  remotes/origin/HEAD   -> origin/master
  remotes/origin/master 539358f update file
  remotes/origin/test   8391dbd update
  remotes/origin/test2  8391dbd update
```

If you had used the -rv option instead, you would not have seen the first line with the local master branch.

There is one more variation you can use with the branch command to find out additional information: the -vv flag. (Note that this is two *v*'s side by side, not one *w*.) The two *v*'s tell Git to show extra information that is very useful—namely, any tracking connections between local branches and the remote tracking (upstream) branches.

```
$ git branch -vv
* master 539358f [origin/master] update file
```

This example is telling you that your local *master* branch is set up to track *origin/master*. You could also use an -a or -r option with the -vv option. However, because those options would only add the same remote tracking branch list, no additional information would be output.

One other way to get the list of branches indirectly is to use the show option for your remote.

```
$ git remote show origin
* remote origin
  Fetch URL: git@diyvb:repos/remote_demo
  Push  URL: git@diyvb:repos/remote_demo
  HEAD branch: master
  Remote branches:
    master     tracked
    test       tracked
    test2      tracked
  Local branch configured for 'git pull':
    master merges with remote master
  Local ref configured for 'git push':
    master pushes to master (up to date)
```

At the bottom of the output, you can see that the local branch master is configured to work with the remote branch master.

Finally, if you take a look at the local config file, you can see where this configuration information is actually stored.

```
$ cat .git/config
[core]
      repositoryformatversion = 0
      filemode = true
      bare = false
      logallrefupdates = true
[remote "origin"]
      url = git@diyvb:repos/remote_demo
      fetch = +refs/heads/*:refs/remotes/origin/*
[branch "master"]
      remote = origin
      merge = refs/heads/master
```

Note the last section about branch master and the resemblance to the data from the git remote show output.

Now that you understand how to view the upstream and tracking branch information, let's look at how to set up the upstream and tracking relationship for branches that don't already have such a relationship.

Configuring Upstream Relationships for Branches

When I talk about *upstream* in Git as it relates to branches, I'm referring to which branch in the remote repository should correspond to a given branch in the local repository. Basically, if you're going to do any of the update operations, such as fetch, pull, or push, between a branch in the local repository and a branch in the remote repository, Git needs to know how to map between the two repositories.

As I discussed earlier, when you initially clone a repository down, you get remote-tracking branches that correspond to the remote branches in the remote repository. And, unless you have specifically cloned the repository as a *bare* repository (meaning one that is not intended for local use), you will also have a default local branch available for you to use. This default local branch (the current

HEAD, usually master) will already be mapped to the corresponding remote tracking branch (for example, master is connected with origin/master). Thus, origin/master is the *upstream* of master.

Automatic Mapping

For other remote tracking branches, if you attempt to start working with a local branch with the same name, Git may establish the tracking relationship for you—or it may not. If you do the traditional git branch command to create a local branch with the same name as one of the remote tracking branches, Git happily allows you to do that, but it does not set up the upstream tracking for you. If, on the other hand, you begin by using a git checkout command to start working with a local branch that has the same name as a remote tracking branch, Git creates the local branch for you *and* establishes the upstream tracking connection. Take a look at the following example to see how this works:

```
$ git branch -av
* master                539358f update file
  remotes/origin/HEAD   -> origin/master
  remotes/origin/master 539358f update file
  remotes/origin/test   8391dbd update
  remotes/origin/test2  8391dbd update

$ git branch test
```

Notice that, at the start, you have the master, test, and test2 remote-tracking branches that were pulled into your clone from the remote. You then create a new branch named *test* using the branch command.

```
$ git branch -vv
* master 539358f [origin/master] update file
  test   539358f update file
$ git checkout test
Switched to branch 'test'
```

Taking a look at the branch -vv output, you see that while *master* is set up to track origin/master, *test* is not.

```
$ git checkout test2
Branch test2 set up to track remote branch test2 from origin.
Switched to a new branch 'test2'
$ git branch -vv
  master 539358f [origin/master] update file
  test   539358f update file
* test2  8391dbd [origin/test2] update
```

You then do a checkout command for a *test2* branch. This time, because you did the checkout instead of the branch command, and because there is a remote-tracking branch with the corresponding name, Git creates the new local branch and sets up the upstream tracking for you. The branch -vv option confirms this for you.

> **NOTE** *You can configure Git so that it defaults to setting up the upstream tracking in these situations, by setting the configuration value for* branch.autoSetupMerge. *Setting this value to* always *tells these commands to set up the upstream tracking by default.*

Manual Mapping

You may have a situation where you create a local branch that does not have automatic upstream tracking done for it. In this case, it is up to the user to explicitly set up the upstream tracking using one of the following methods.

Git provides a way to specify what the upstream relationship should be when a branch is created. By default, if you choose an upstream branch as the starting point (meaning you supply it as the second branch in the following format), Git sets up the tracking.

```
$ git branch test origin/test
Branch test set up to track remote branch test from origin.
```

The same result occurs if you explicitly include the --track option.

```
$ git branch --track test origin/test
Branch test set up to track remote branch test from origin.
$ git branch -vv | grep test
  test   8391dbd [origin/test] update
```

In the previous note, I mentioned the *autoSetupMerge* configuration value. Having this value set implies --track by default. Similarly, if for some reason, you don't want to have the upstream tracking set up, you can supply the --no-track option.

```
$ git branch --no-track test origin/test
diyuser@diyvb:~/remote_demo4$ git branch -vv | grep test
  test   8391dbd update
```

Notice the absence of the upstream tracking branch information in the branch command output.

There is also an option named --set-upstream, which is the same as --track in most cases. However, this option is deprecated in Git and will be unsupported at some point, so it is best not to use it. As of this writing, if you use it, it will still work, but you will receive a message that it is being deprecated.

```
$ git branch --set-upstream test2 origin/test2
The --set-upstream flag is deprecated and will be removed. Consider using --track
or --set-upstream-to
Branch test2 set up to track remote branch test2 from origin.
```

The other available option is the newer *--set-upstream-to* option. This option performs a similar function to *--track* and *--set-upstream* but is clearer.

```
$ git branch test
$ git branch -vv | grep test
  test   539358f update file
$ git branch --set-upstream-to=origin/test test
Branch test set up to track remote branch test from origin.
$ git branch -vv | grep test
  test   539358f [origin/test: ahead 4, behind 1] update file
```

Notice that when using the --set-upstream-to option, you set it to the desired value with an equals sign and pass the local branch after the upstream assignment. This option can be abbreviated as *-u*. Also note that you can set the upstream-tracking branch to any of the branches, not just the one that corresponds in name. In the following example, I set the upstream to be origin/test2 for the local test branch (instead of origin/test).

```
$ git branch test
$ git branch -vv | grep test
  test   539358f update file
$ git branch -u origin/test2 test
Branch test set up to track remote branch test2 from origin.
$ git branch -vv | grep test
  test   539358f [origin/test2: ahead 4, behind 1] update file
```

Next, you'll look at the push operation and see how the interaction between the different types of branches is used with it.

Push

As you saw in my earlier promotion model for Git, you can use the push command to update the remote repository. Basically, you can think of it as trying to take changes you've committed into your local repository and mirror them to the remote repository. As part of the same operation, push takes note of which commits the corresponding branches in the remote repository point to, and it updates the remote tracking branches in the local repository to point to the same mirrored versions there. The syntax for the push command is as follows:

```
git push [--all | --mirror | --tags] [--follow-tags] [--atomic] [-n | --dry-
run] [--receive-pack=<git-receive-pack>]
         [--repo=<repository>] [-f | --force] [-d | --delete] [--prune] [-v |
--verbose]
          [-u | --set-upstream]
          [--[no-]signed|--sign=(true|false|if-asked)]
          [--force-with-lease[=<refname>[:<expect>]]]
          [--no-verify] [<repository> [<refspec>...]]
```

Let's look at an example of what a push operation would look like between a local repository and a remote repository. In this case, you'll mimic the idea of pushing to an empty remote repository from an existing local repository.

> **TIP** *If you are creating a new project or you want to share an existing project with others, you may want to create a new remote repository from your existing code base in your local repository. There are a couple of ways to do this, depending on your particular circumstances.*
>
> *A prerequisite here is that you must have permissions where you want to create the remote repository from your local one. Once you have permissions in place, there are essentially three ways you can create a remote repository directly from your local repository:*
>
> - *You can initialize a new repository here with the* git init *command and push your local repository to it.*
> - *You can create a bare version of your local repository and copy it to the desired location.*
> - *If you are working in some other kind of interface (such as a hosting service like GitHub), they will have their own way for users to create a new repository area.*

Since the last option is provider-specific, we'll look at the first two options here.

Approach #1: Using git init

1. In the remote area for repositories, make a directory for the repository, change into that directory, and run the following command:

   ```
   $ git init --bare --shared<options>
   ```

   The --shared option isn't required, but it does allow you to configure permissions and access at creation time. See the git init --help page for an explanation of --shared and its possible values.

2. Back in your local environment, add the URL (path) for the new remote area you created as origin (or whatever remote reference name you'd like) to your repository.

   ```
   $ git remote add origin http://mygitserver.com/myrepopath
   ```

3. Now you have a connection to an empty remote. You can use the push operation to push your content over to the remote (which I'll talk about shortly) using commands like this:

   ```
   $ git push origin master
   ```

Approach #2: Copying a bare repository

If you have write access to the area where the remote repositories are hosted, you can create a bare repository and copy it there. The steps are pretty straightforward.

1. Create a bare repository from your local repository. This creates a copy of the repository that is suitable for hosting as a remote repository.

   ```
   $ git clone --bare myrepo myrepo.git
   ```

2. Copy the bare version of the repository out to the remote repository area. In this example, I use the scp command.

   ```
   $ scp -r myrepo.git  user@server.com:/git/area/for/repos
   ```

3. Of course, with either of these methods, you will need to update any other permissions or access protocols to ensure that the appropriate users can then clone the new repositories.

Figure 12-5 shows a local repository with three commits that have been made into it and an empty remote repository.

If you now do a push operation, Git takes your local changes and updates the remote repository with them (see Figure 12-6).

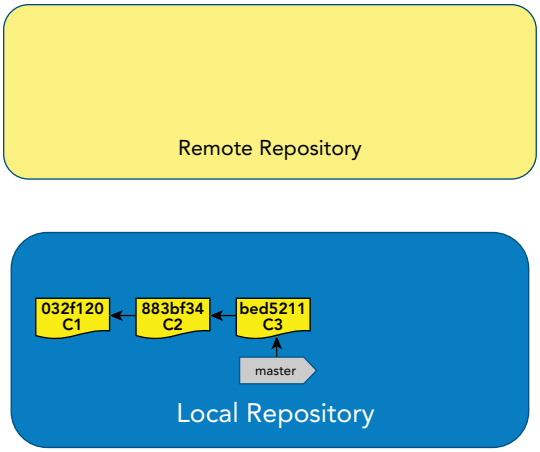


FIGURE 12-5: Initial changes in the local repository

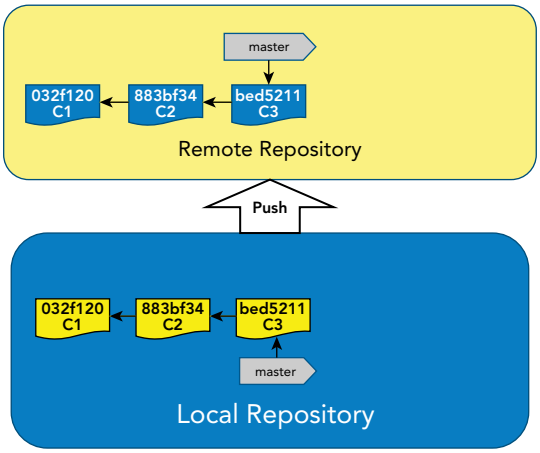


FIGURE 12-6: After a push to the remote repository

Git also creates a branch pointer in the local repository to indicate where the corresponding branch in the remote repository is pointing (see Figure 12-7). This creates the remote tracking branch that I have been talking about. This branch has the namespace of the remote reference (*origin* by default), so you can reference it as *origin/master*. You can now reference the origin/master branch as a representation of the master branch on the remote without being connected to the remote.

Seeing Status: Local versus Remote

Carrying this sequence one step further, suppose you now make a commit into the local repository that is not (yet) pushed over to the remote repository, as shown in Figure 12-8.

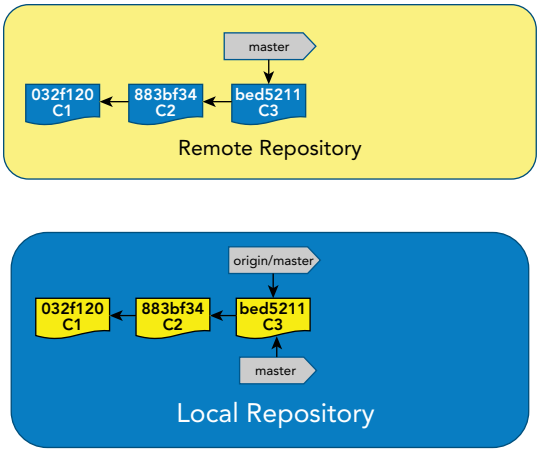


FIGURE 12-7: Remote tracking branch created in the local repository

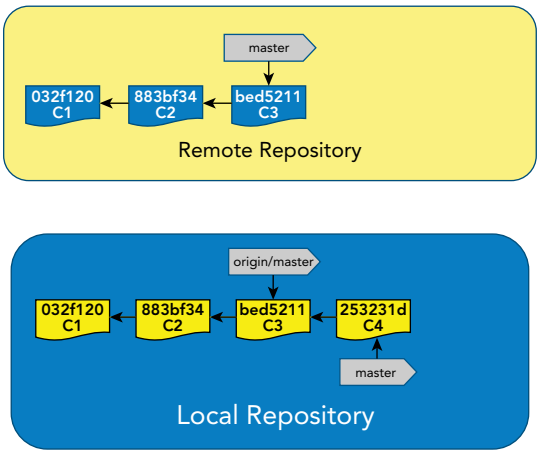


FIGURE 12-8: After a commit into the local repository

The interesting thing about this state is what a *git status* command would show.

```
$ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)
nothing to commit, working directory clean
```

Notice the wording here. You can read the first sentence as: "Compared to the last time you checked in with the remote, your local branch has one newer commit on it." This is because origin/master

(the remote tracking branch) represents where master was the last time you communicated with the remote. To see the differences another way, you can use the *git branch -av* command.

```
$ git branch -av
* master                  253231d [ahead 1] <commit message>
  remotes/origin/master   bed5211 <previous commit message>
```

Notice the shorthand notation here, *[ahead 1]*, which means the same thing as "master is ahead of origin/master by one commit."

You can try one other variation by using the *-vv* option.

```
$ git branch -vv
* master   253231d [origin/master: ahead 1] <commit message>
```

Push Formats

The push command expects a repository and a reference to push to. For your purposes here, the repository will always be a remote reference name (such as *origin*) and the reference will always be one or two branch names. The reason I say *one or two* is because, while you typically only specify the branch you are pushing to, it is also possible to specify a different source branch to push from (see the following examples).

So, in most cases, you would use this form:

```
$ git push <remote repository> <remote branch>
```

You can, however, also use this form if you are pushing from a local branch that is named differently from the targeted remote branch:

```
$ git push <remote repository> <local branch>:<remote branch>
```

There is also a default form of the push command, which is just

```
$ git push
```

In the default form, the repository defaults to *origin*. However, the default branch behavior is configurable and requires some additional explanation, as I discuss in the section, "Understanding the Push Default Behavior."

```
$ git push origin :
```

In this form, the command pushes all matching branches. For a description of what *matching* means here, see the section "Understanding the Push Default Behavior."

```
$ git push origin HEAD
```

This is the same form as I described earlier, but it's worth mentioning because it provides a convenient way to just tell Git to push the current branch (in this case, HEAD) to the branch of the same name on the remote.

Finally, it's worth noting that you can push multiple branches on the command line or multiple local:remote pairs.

```
$ git push origin branch1 branch2 branch3
$ git push origin lbranch1:rbranch1 lbranch2:rbranch2 lbranch3:rbranch3
```

> **NOTE** *If you create a new local branch and then want to create it in the remote (assuming it doesn't already exist), the process is just to push it, as follows:*
>
> ```
> $ git push <remote> <branchname>
> ```
>
> *The formal syntax is*
>
> ```
> $ git push <remote> <local branchname>:<remote branchname>
> ```
>
> *However, Git assumes that if you omit <remote branchname>, it should just be the same.*

Understanding the Push Default Behavior

The reason you have different options for the push command's default behavior has to do with how you map branches from the local repository to branches in the remote repository. If no reference (branch) is specified directly on the command line, is already configured, or is implied by a command line option, Git needs to know what to do in terms of which local branch to push to which remote branch.

To specify your choice for the default value, you configure the Git configuration setting *push.default*. The possible values are as follows:

- *nothing*—Don't push anything.
- *matching*—Push all the matching branches. *Matching* here means that the names match between a local branch and a remote branch.
- *upstream*—Push the current branch to the one defined as its upstream branch. (This was formerly known as tracking.)
- *current*—Push the current branch to the branch on the remote side with the same name.
- *simple*—Like upstream, but don't allow the push if the upstream's branch name is different from the local branch's name. (This is the default value as of Git 2.0.)

If you're only concerned with pushing to a single branch at a time, or if you want more control, you can use the *simple*, *current*, and *upstream* modes. If you want to push all corresponding branches, you can use *matching*.

Upstream, in the context of a Git remote environment, generally refers to the location of the remote repository, but it has a lot of context around it when setting up branches. I try to clarify that context in the earlier section, "Configuring Upstream Relationships for Branches."

The simple, current, and upstream modes are for when you want to push out content from a single branch, even when the other branches are not yet ready to be pushed out.

Push Options

Push has many options that you can use to customize its behavior. I'll cover a few of the most common and significant ones here, but you can find more information online on the push help page.

all

The all option pushes all branches. An example would be:

```
$ git push --all
```

delete

The delete option deletes the reference on the remote repository. For example, the following command deletes a remote branch:

```
$ git push --delete origin testing
- [deleted]         testing
```

You can also use the colon (:) notation to tell Git to delete something. For example, this command has the same effect as the previous one.

```
$ git push origin :testing
- [deleted]         testing
```

The delete option is a global option to push that applies to all references specified on the command line. The colon (:) allows for specifying particular references to delete if there is a list.

> **WARNING** *The colon (:) is used both to signify a branch to delete and to separate a local branch from the remote branch in a push context. So, you should be careful when using it with a push command. For example, the following syntax pushes the contents of the* testing *local branch to the* testing2 *remote branch:*
>
> ```
> $ git push origin testing:testing2
> Counting objects: 3, done.
> Writing objects: 100% (3/3), 249 bytes | 0 bytes/s, done.
> Total 3 (delta 0), reused 0 (delta 0)
> To C:/Program Files/Git/force_repo.git
>  35ab623..6b87658  testing -> testing2
> ```
>
> *However, if you accidentally include a space, the command does something completely different.*
>
> ```
> $ git push origin testing :testing2
> Counting objects: 3, done.
> Writing objects: 100% (3/3), 242 bytes | 0 bytes/s, done.
> Total 3 (delta 0), reused 0 (delta 0)
> To C:/Program Files/Git/force_repo.git
>  35ab623..e42f850  testing -> testing
>  - [deleted]         testing2
> ```

tags, follow-tags

By default, push does not push tags over to the remote side. The option *--tags* tells Git to push tags. The --follow-tags option is more refined and pushes annotated tags that are reachable in the current commits and missing in the remote.

force

Git typically refuses to accept a push to the remote unless it can do a fast-forward merge—meaning the changes being pushed already contain the latest contents of the remote branch and no one else has made intervening changes. Here's an example of what Git returns in this case:

```
$ git push
To C:/Program Files/Git/./calc2.git
 ! [rejected]        master -> master (fetch first)
error: failed to push some refs to 'C:/Program Files/Git/./calc2.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

I talk more about the best way to handle this situation in Chapter 13. However, there is a --force option (short version -f) that you can use to force the push to happen in these cases. This can be dangerous, though, and you should only use it if you truly understand the consequences.

> **WARNING** *If another user has pushed updated content and you force your changes in without pulling the updates first, you can end up* orphaning *their changes.*
>
> *From that user's perspective, after their push, their changes are at the head of the branch. When you force your changes in, your changes are appended at the point the branch was the last time you did an operation on the remote (where your remote tracking branch points). This is likely the same place their changes were appended.*
>
> *Once your changes are pushed, they become the new head and the other user's changes are effectively orphaned off to the side. Eventually, their changes may be removed completely by garbage collection.*

You can use the force option as follows:

```
$ git push -f origin master
Counting objects: 10, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (10/10), 856 bytes | 0 bytes/s, done.
Total 10 (delta 0), reused 0 (delta 0)
To C:/Program Files/Git/./calc2.git
 + 0caf33c...7f3fb23 master -> master (forced update)
```

Notice the "forced update" option message at the end.

Git also allows you to use a plus (+) sign in front of the branch argument to force an update.

```
$ git push origin +master
Counting objects: 14, done.
```

```
Delta compression using up to 4 threads.
Compressing objects: 100% (8/8), done.
Writing objects: 100% (14/14), 1.14 KiB | 0 bytes/s, done.
Total 14 (delta 0), reused 0 (delta 0)
To C:/Program Files/Git/./calc2.git
 + 7f3fb23...13f956a master -> master (forced update)
```

The difference in using the --force option and the plus sign is that the --force option is a global option to push and so applies to all branches being pushed. The plus sign is applied on a branch-by-branch basis as desired.

mirror

The mirror option tells Git to push everything (tags, remotes, heads, and so on) over. You typically use the --mirror option if you need to migrate or move an entire repository somewhere.

set-upstream

For every branch that is successfully pushed by the push operation or that is up-to-date, the set-upstream option adds an upstream tracking reference. This may be needed on initial pushes for branches that do not already have an upstream tracking reference. By setting the upstream tracking reference, you will also help to ensure that any subsequent operations that use this reference (such as a git pull) will know what to do. The short version of this option is *-u*.

tags

The tags option tells Git to push over all of the tags (all of the items under ref/tags in the .git directory). You need to specify this option to ensure tags are pushed along with the other content.

Now, let's look at how you can update the local environment with changes from the remote repository using the fetch and pull commands.

Fetch

Once a repository is cloned, the connection between the remote branches and remote tracking branches in the local repository is established. The fetch command allows you to get the latest updates to the remote tracking branches (as well as tags). The syntax is fairly straightforward.

```
git fetch [<options>] [<repository> [<refspec>...]]
git fetch [<options>] <group>
git fetch --multiple [<options>] [(<repository> | <group>)...]
git fetch --all [<options>]
```

To understand the basic mechanics of a fetch operation, take a look at Figure 12-9. This is a continuation of the set of commits you were using in the push discussion (shown in Figures 12-5 to 12-8).

On the left side of Figure 12-9 is your starting point. You've pushed the most recent commit you made locally over to the remote. However, someone else has also made a change and pushed it into the remote repository. Thus, you have an additional commit in the remote repository that you don't have in your local repository.

If you were to do a git status command at this point, you would get the following output:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

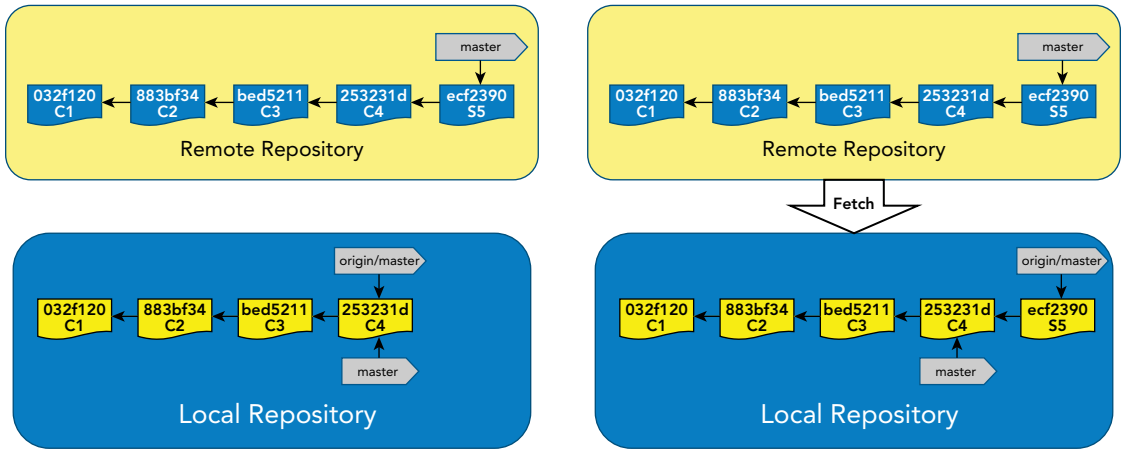


FIGURE 12-9: Before and after a fetch operation

This may seem confusing at first because I just said that there was newer content in the remote repository that you don't have in your local repository. However, remember that you're using the remote tracking branch (origin/master) as your bookmark of where the corresponding branch on the remote is. And the remote tracking branch is only updated when you check in with the remote. So, based on the last time you checked in with the remote, both the remote master branch and your local master branch were pointing at the same commit.

Now, if you look at the right side of Figure 12-9, you see a fetch operation being performed. The fetch operation communicates with the remote repository to see if there are any updates there that it needs to reflect in the corresponding remote tracking branch. There are: there is a new commit on master in the remote repository, as I previously noted. So, Git updates the repository with the new commit and (only) moves the remote tracking branch (origin/master) to point to the new commit. Notice that your local branch, master, is not updated. Fetch only updates the contents of the repository and remote tracking branches.

After the fetch operation, if you check the status, you see something like this:

```
$ git status
On branch master
Your branch is behind 'origin/master' by 1 commit, and can be fast-forwarded.
  (use "git pull" to update your local branch)
nothing to commit, working directory clean
```

This tells you that your current branch master is one commit behind where the remote tracking branch is. (Another way to say this is that the remote tracking branch, and thus the remote, has one newer commit that your local branch is not yet aware of.) Notice that the output also tells you that your local branch can be fast-forwarded. I'll discuss what that means shortly.

Fetch Options

As with the push command, there are a few options you should know about for the fetch command.

depth

The depth option limits fetching to the number of commits specified as an argument to the option. For example, --depth=1 only fetches down the latest changes (latest commit).

force

Like push, fetch also has a force option. If you attempt to fetch from a branch on the remote repository to a remote tracking branch in the local repository and Git can't do a fast-forward merge, then it refuses to do the update. However, you can use the --force (-f) option to force Git to do the update. Of course, you should only do this if you understand the consequences and are sure you want to do it.

To illustrate this, I'll use an example of trying to fetch updates from one branch into a completely different branch. Attempting this when Git can't do a fast-forward merge between the two branches results in an error message like the following one:

```
$ git fetch origin master:ui
! [rejected]        master     -> ui  (non-fast-forward)
```

You can use the -f option to force the update to happen.

```
$ git fetch -f origin master:ui
+ fbcf72c...8a3f48b master     -> ui  (forced update)
```

As an optional method of forcing the update, you can supply a plus (+) sign in front of the branches.

```
$ git fetch origin +master:ui
+ abaf124...8a3f48b master     -> ui  (forced update)
```

The -f (or --force) option is an option to fetch, and it applies to all branches passed to the command. You can use the plus (+) sign to specify forcing any particular branch if you don't want to do all of them.

Synchronizing Local Branches after a Fetch

Recall that in an earlier example, after fetching the latest updates, the status said that your local branch was one commit behind the remote tracking branch and could be "fast-forwarded."

This means that a fast-forward merge can be done to bring the local branch up-to-date with the latest changes to the remote tracking branch (and thus the remote branch). The left side of Figure 12-10 shows the local repository after the fetch but before the merge. Note that the master branch is one commit behind the remote tracking branch.

You can then do the merge.

```
$ git merge origin/master
Updating 253231d..ecf2390
Fast-forward
 ...
```

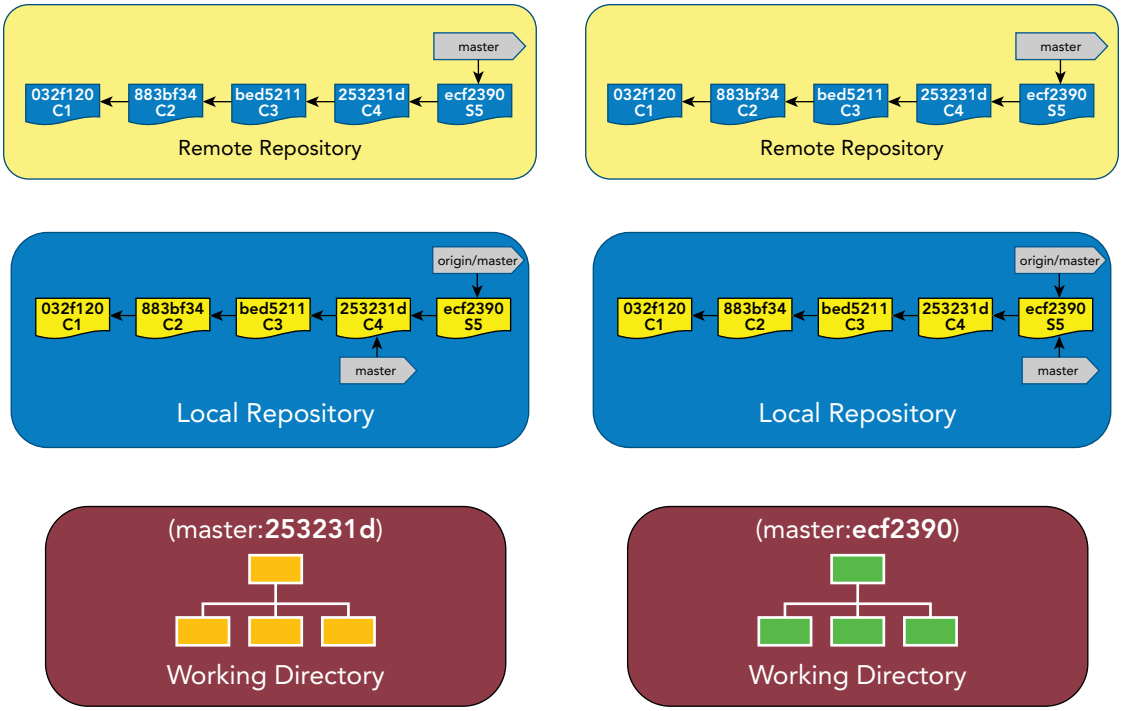


FIGURE 12-10: The local repository before and after the merge

You end up with the results on the right side of Figure 12-10. Notice that the master branch has been fast-forwarded to the same commit as the remote-tracking branch. Also, as I discuss in Chapter 9, the merge occurs in the working directory as well as the local repository. This is shown in Figure 12-10 by the change in contents in the working directory before and after the merge.

This is how you synchronize a local branch after a fetch updates. After the merge, your local repository, local branch, and files in the working directory all have the latest updates from the remote.

Pull

If you understand fetch and merge, then you understand the pull command. In its default form, the git pull command is essentially a fetch followed by a merge. Figure 12-11 represents a before-and-after view that is similar to the other commands.

The syntax for the pull command is also fairly straightforward:

```
git pull [options] [<repository> [<refspec>...]]
```

Let's look at a few of the options for the pull command.

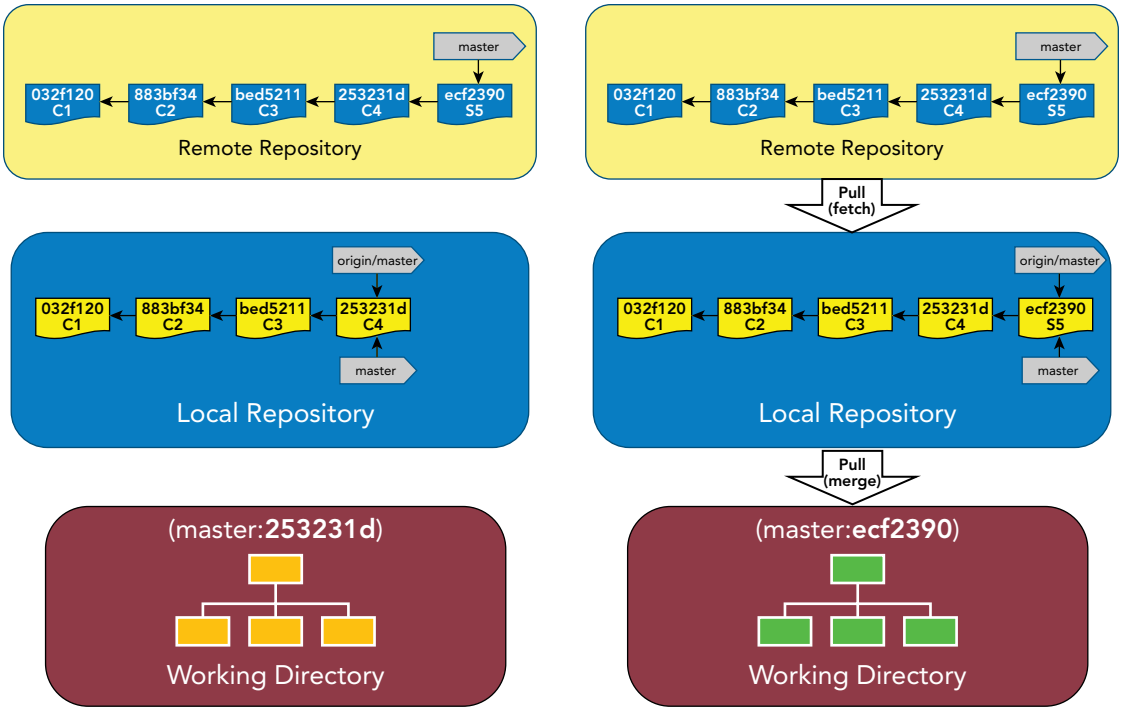


FIGURE 12-11: Before and after a pull operation

Pull Options

Because the pull command is basically a combination of a fetch and merge, pull has a set of options that correspond to fetching and a set that correspond to merging. For extended information about merging or rebasing, refer to Chapter 9.

Options Related to Merging

Following are a few useful options related to merging that are supported by the pull operation.

no-commit

The no-commit option performs the merge but does not commit the results. It provides an opportunity to inspect the results of the merge before committing.

no-edit

The no-edit option tells Git not to invoke the editor before the commit—just to accept the automatically generated message.

rebase

The rebase option has several possible settings. The main idea is that instead of just doing a fetch and merge for the pull action, Git does a fetch and rebase. Beyond that, you can use other settings:

- =true—After the fetch, this setting rebases the current branch on top of the upstream branch.

- =false—This setting merges the current branch into the upstream branch.
- =preserve—This setting rebases with the rebase operation's --preserve-merges option. This forces Git to also use merge commits that are part of the history as part of the rebase.
- =interactive—This setting starts up the interactive mode of rebase.

strategy

The strategy option supplies a merge strategy to use (-s for short). Note that this option can be supplied more than once. If multiple instances are supplied, that indicates the order in which Git should try to use each strategy.

strategy-option

Specifying this option to pull allows you to supply a particular option to the chosen merge strategy (-X for short).

Options Related to Fetching

The other component of the pull operation has to do with fetching. Here are a few useful options related to that aspect.

depth

Like cloning, the depth option does a shallow pull where the history is truncated to the number of commits that you specify as an argument to the option. For example, --depth=1 only pulls down the latest changes (one commit).

One interesting point about this option for the pull command is that it can lengthen or shorten the depth that was previously used. For example, if you clone down a repository with a depth of 1, but later decide you want a depth of 3, you can use the pull command with that depth.

force

Like push and fetch, the pull command also has a force option. If you attempt to pull from a branch on the remote repository to a branch in the local repository and Git can't do a fast-forward merge, then it refuses to do the update. However, you can use the --force (-f) option to force Git to do the update. Of course, you should only do this if you understand the consequences and are sure you want to do it.

To illustrate this, I'll again use an example of trying to fetch updates from one branch into a completely different branch. Attempting this when Git can't do a fast-forward merge between the two branches results in an error message like this:

```
$ git pull origin features:docs
From https://github.com/brentlaster/calc2
 ! [rejected]        features   -> docs  (non-fast-forward)
```

You can use the -f option to force the update to happen.

```
$ git pull -f origin features:docs
From https://github.com/brentlaster/calc2
 + abaf124...b372aa6 features   -> docs  (forced update)
```

As an optional method of forcing the update, you can supply a plus (+) sign in front of the branches.

```
$ git pull origin +features:docs
From https://github.com/brentlaster/calc2
 + abaf124...b372aa6 features   -> docs  (forced update)
```

The -f (or --force) option is an option to fetch, and it applies to all branches that are passed to the command. You can use the plus (+) sign to specify forcing any particular branch if you don't want to do all of them.

SUMMARY

In this chapter, I switched to the other side of the Git environment: remotes. I clarified what the term *remote* can mean in Git—from an alias for the longer URL path to the remote repository, to the remote repository itself.

I talked about the various networking protocols (SSH, Git, Local, and HTTP—both *smart* and *dumb*) that you can use with Git to communicate between the local and remote sides.

I described in more detail how you clone down a copy of the remote repository to create a new local environment. I also explored some of the options that are available to limit or otherwise modify the set of content that you clone.

I discussed the differences between remote branches, remote tracking branches, and local branches, and I described how these branches work in practice. I showed you how to get a list of these branches and how Git tells you if you are behind or ahead locally, compared to the remote.

Finally, I discussed the main operations that you use to interact with the remote side: push, fetch, and pull. I explored how each of these operations affects the local environment, as well as some of the key options that you are likely to use with each one.

In the next chapter, I continue discussing remotes by looking at the typical workflow you can use with remotes, and exploring an alternative workflow that is popular for hosted sites.

About Connected Lab 8: Setting Up a GitHub Account and Cloning a Repository

This lab is intended to give you some practice in working with remotes. To do this, you'll set up a GitHub repository and see how some of the basic concepts such as forking, cloning, and adding remote references work in practice.

Connected Lab 8

Setting Up a GitHub Account and Cloning a Repository

In this lab, you'll get some practice with remotes by setting up a GitHub account, forking a repository, and cloning it down to your system to work with.

PREREQUISITES

This lab requires that you have Internet access.

STEPS

1. Go to `https://github.com`.
2. Fill in the *Pick a username*, *Your email address*, and *Create a password* fields.
3. Click the *Sign up for GitHub* button.
4. Accept the defaults on the next screen and click the Continue button.
5. (Optional) Fill out the Interests sections on the next page or just click the *skip this step* link.
6. Follow the instructions to verify your email address. Then click the Start a project button.
7. Go to the calc2 project at `https://github.com/professional-git/calc2.`
8. Click the Fork button (top-right corner) so that the repository is forked to your user ID. (Your URL changes to `https://github.com/<github userid>/calc2.)`
9. Open up a terminal session (that you can run Git in) on your local machine, and change back to your home directory (or at least out of any directories that have Git repositories in them).

   ```
   $ cd ~
   ```

10. On the right side of the calc2 project web page, click the *Clone or download* button. A pop-up window appears, populated with the URL path you can use to clone this project down using the HTTPS protocol. To the right of that URL, click the clipboard icon to copy the path to your clipboard. This saves you from having to construct the path yourself.

11. Switch back to your terminal session. Clone the project down by typing **git clone** and then pasting the path from the clipboard. Press Enter.

```
$ git clone https://github.com/<your github user id>/calc2.git
```

You see some messages from the remote side, and then the project is cloned down into the calc2 directory.

12. You are now going to create another cloned copy. This copy will emulate another person working in the same repository and allow you to see what happens in the next lab when someone else makes a change to the same code base that you're working on. When you already have a copy of a repository on your disk, you can create another copy by supplying a new name for the destination directory to the command. You'll use *calc_other* here. From the same directory as before, run the second clone command.

```
$ git clone https://github.com/<your github user id>/calc2.git calc_other calc_
other
```

13. You can now browse around the calc2 and calc_other directories. Each directory only contains one file, but if you look at the hidden files, you can see the .git repository that was cloned down from the remote. You can also run commands like *branch* to see the set of branches in each cloned repository. Try the following commands to see the list of remote branches and information about the most recent set of changes in each cloned repository. (Note that you will need to change into the respective directory for the repository first.)

```
$ git branch -r
$ git branch -av
```

14. You will now do one more operation for both repositories. Assume that you want to be able to pull in any updates that may be made in the original GitHub repository that these repositories are forked from. In this case, you need to add another remote that you can run pull or fetch operations against if you want to. In each of the directories (calc2 and calc_other), run the following commands, changing into the respective directory for each cloned repository first:

```
$ git remote add upstream https://github.com/professional-git/calc2.git
```

15. Run the remote -v operation to see the set of remotes on each area.

```
$ git remote -v
```

13

Understanding Remotes—Workflows for Changes

WHAT'S IN THIS CHAPTER?

- Dealing with changes from multiple users
- Resolving conflicts after updating from a remote
- Rebasing versus merging after a pull
- Using the fork-and-pull model for collaboration and contribution
- Stashing, a better way to merge
- Tying all the workflow pieces together

So far in this book, I've focused mostly on working with Git as an individual user in a local environment. In Chapter 12, I extended this discussion to interacting with the remote environment. In this chapter, I'll focus on working with other users.

I'll first explore the basic conflict-merge resolution workflow that comes into play when someone else has changed code you've also been working on. I'll show you how this workflow works and explain why the process happens the way it does. Next, I'll present a modified workflow that is widely used for contributing to other users' projects. Finally I'll discuss a strategy to help mitigate surprises when you pull updates, and fit that strategy into a larger workflow that is similar to what you might use with some other source control systems.

NOTE Workflows *is a term that is often overused in technical documents. With Git, I previously talked about adding and committing content in the local environment. The workflows that I discuss in this chapter extend that model further—to interactions and strategies when pushing changes to the remote repository, and working in an environment where others are also pushing changes to the same codebase.*

Another common type of workflow in Git has to do with branching workflows, or more precisely, branching strategies, where you use multiple branches to manage the flow of changes over time. I discuss those ideas in Chapter 8. However, as I note in that chapter, you can find more information and strategy suggestions on the web. In particular, you can search for git flow *and find a wealth of content (and opinions) about various strategies.*

One other source of information related to branching workflows is the built-in help page for workflows *that comes with Git. This information is subtitled "An overview of recommended workflows with Git." This is not a command, but a help topic, so you can access it by executing the command*

```
$ git workflows --help
```

or

```
$ git help workflows
```

THE BASIC CONFLICT AND MERGE RESOLUTION WORKFLOW IN GIT

Your interaction with other users in Git all comes together when you start interacting with remote repositories. Whether pushing or pulling, or fetching and merging, updating content to and from the remote side gives you your first indication of whether there are conflicts between the changes you've made and the changes others have made.

How the Remote Side Handles Conflicts

Recall that I said in Chapter 12 that remotes are lazy. Basically, the remote side expects that whatever you are trying to push from the local side already includes all of the content that is currently in the destination branches in the remote repository. The remote side is expected to be an ancestor of what you're trying to push; you're just adding on at the end. This is so that Git can simply do a fast-forward merge, which means that no actual merging has to occur on the remote.

If you were the only one working in a remote repository, this usually wouldn't present a problem. However, remote repositories are primarily meant to provide a place for multiple users to push their changes and share code. So, at some point, you will probably encounter a situation where Git cannot do a fast-forward merge to incorporate your changes. This occurs when someone else has pushed updates (into the remote) that you haven't yet pulled and merged into your pending changes.

Put another way, someone beat you to the punch, getting their changes in on top of the same code base that you were working on. Because updates have been pushed that potentially conflict with

updates that you're trying to push, the remote side of Git just flags this conflict and stops the operation, rejecting your push. It's then up to you to sort the conflict resolution out in the local environment and try again.

NOTE *You will be rejected by Git at some point. In this case, rejection refers to the way Git tells you it can't do a fast-forward of your changes in the remote repository. The typical message looks something like this:*

```
$ git push
To C:/Program Files/Git/./calc2.git
 ! [rejected]        master -> master (fetch first)
error: failed to push some refs to 'C:/Program Files/Git/./calc2.git'
hint: Updates were rejected because the remote contains work that
you do
hint: not have locally. This is usually caused by another repository
pushing
hint: to the same ref. You may want to first integrate the remote
changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for
details.
```

As ominous as this may look, it's not that bad, and in most cases the conflict can be easily resolved, as I discuss in the rest of this chapter.

Conflict Scope

If you've worked with any other source management systems, you're probably familiar with a scenario like the one just mentioned. In those systems, the typical granularity is a file and the system records the delta change (Figure 13-1). The conflict usually occurs when you make a change to a file that someone else has also changed. They check in their change before you do. Then, when you check in your change to the same file, the system tells you there is a merge conflict.

FIGURE 13-1: File granularity corresponding to delta changes

The idea in Git is similar, but the scope is wider. Recall that Git records changes as commits (or *snapshots*) of entire trees with files and directories. As a result, Git operates based on the idea of commits (snapshots) versus files for scope of change (Figure 13-2).

This means that if anything (any file) has been changed by someone else within the scope of the commit (snapshot) since you started making changes, you get a merge conflict—that is, you are rejected—when you go to push your changes over to the remote repository.

The idea you have to get used to is that you can get a merge conflict (inability to do a fast-forward) even if both you and the other user have changed entirely different files, even in different directories.

To illustrate this, consider a situation with two users working in Git. To start with, both users have cloned the remote repository down to their local environment (Figure 13-3).

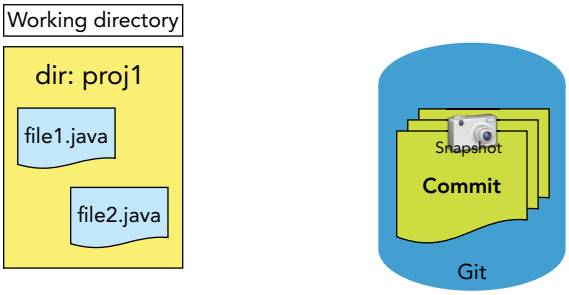


FIGURE 13-2: Commits are a snapshot of files and directories.

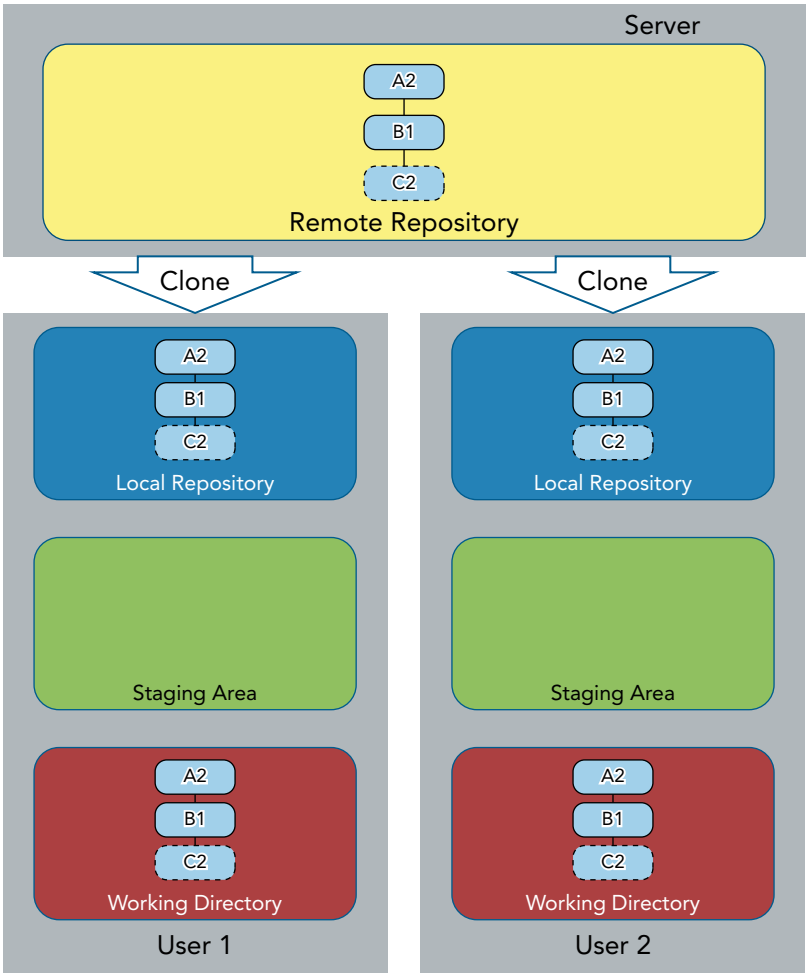


FIGURE 13-3: Two users with the same cloned contents

Now, let's suppose that User 1 modifies files A2 and C2, then stages and commits them. User 1 then pushes those changes back over to the remote repository. Because User 1 is the first of the two users to push their changes in, they can get their changes in without any merge issues, as shown in Figure 13-4.

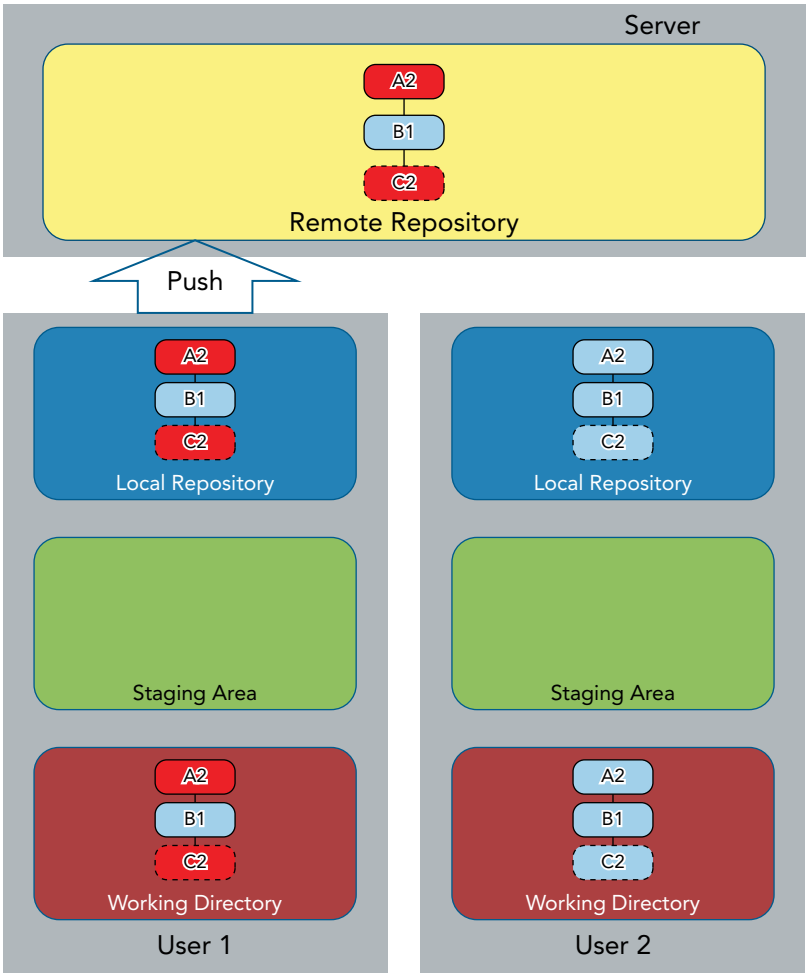


FIGURE 13-4: User 1 successfully pushes their changes.

While this is occurring, User 2 is working on modifications to the same commit in their working directory. However, they are only modifying file B1—the one file that User 1 did not touch. When they are done, they commit the change and attempt to push it into the remote repository. At this point, Git rejects the push (as indicated by the X in Figure 13-5).

When trying to do this push that Git can't fast-forward, the user generally sees a message like this:

```
$ git push
To C:/Program Files/Git/./calc2.git
 ! [rejected]        master -> master (fetch first)
```

```
error: failed to push some refs to 'C:/Program Files/Git/./calc2.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

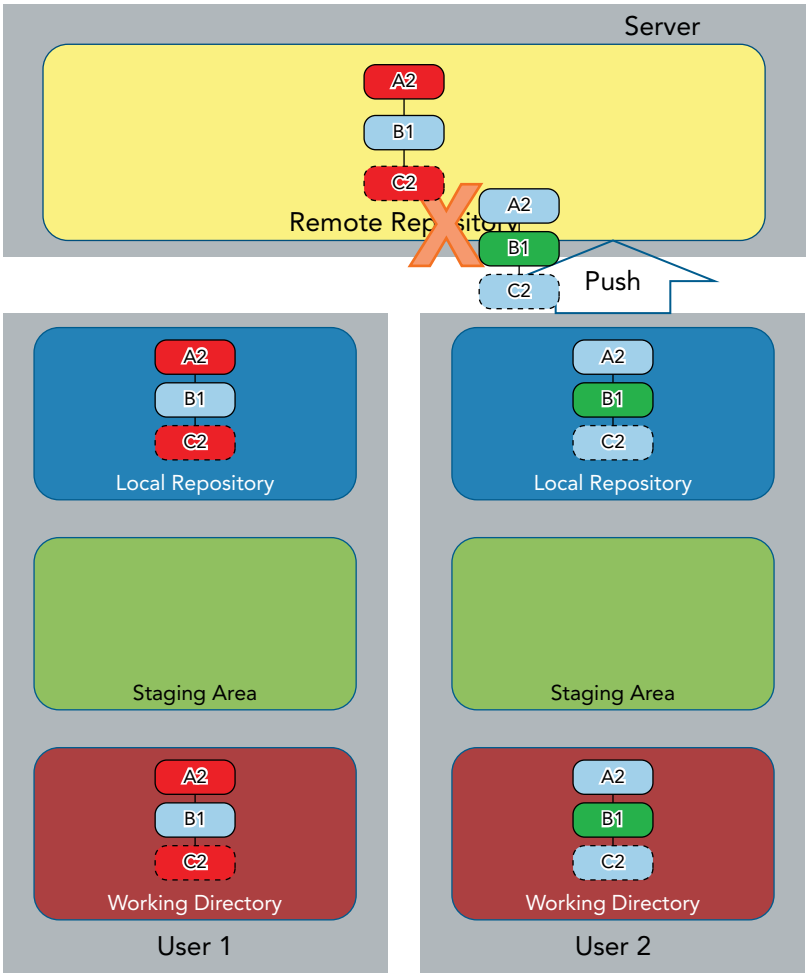


FIGURE 13-5: User 2 attempts to push their changes and is rejected.

Notice that even though User 2 has changed a file that User 1 did not touch (and that User 2 hasn't changed any of the same files that User 1 did touch), Git still sees this as a merge conflict. Again, this is because multiple users changed some content within the same snapshot (commit).

As far as the remote side is now concerned, there is a merge conflict, and it's up to the user to resolve the problem in a local environment and then push the merged content again. (The presumption is then that the merged content can be fast-forwarded.)

Resolving the Conflict

Here's an example of what that resolution process could look like. In Figure 13-6, User 2 has changes they cannot currently push. It is therefore up to them to get the latest changes and merge them locally into their changes. Getting the latest updates into the local environment can be done with a fetch into the repository and then a merge of the remote-tracking branch into the current branch. It can also be done with a pull operation, because the pull operation updates the local repository *and* attempts to merge in the changes in the working directory as well.

> **NOTE** *It is also possible to tell Git to force a non-fast-forward change. You can do this with the --force (-f) option to the push command, or by adding a plus symbol (+) on the front of the branch being pushed to. This would look like either*
>
> ```
> git push --force origin master
> ```
>
> or
>
> ```
> git push origin +master
> ```
>
> *This approach is generally not recommended, though, and can cause non-trivial issues with wiping out other changes (see the "Push" section in Chapter 12). Note that to even allow this, the Git configuration must have the value* receive .denyNonFastForwards *set to* false. *In general, you want this value set to true to prevent non-fast-forward pushes.*
>
> *At times, you may want to set this value to false so that you are able to force a non-fast-forward push. For example, the state of the branch you are pushing to may become substantially incorrect and so you may just want to force an update to correct it again.*

For simplicity, I'll illustrate resolving this conflict in Figure 13-6 with User 2 doing a pull operation.

Once this pull is complete, the local merge completes without any problems because the same files weren't changed. However, what if the same files *were* changed? In that case, more manual merging would probably be required. There are two basic scenarios here: merging after a basic pull operation and merging after a pull with the *--rebase* option. Let's take a look at how each of these merge operations might be used in practice.

Dealing with Merges after a Pull

Suppose you have two users making changes to an instruction file for an application. One user is working on chapter 1 and the other is working on chapter 2. The initial version of the file that was pushed looks like this:

```
$ cat instructions.txt
User Instructions
```

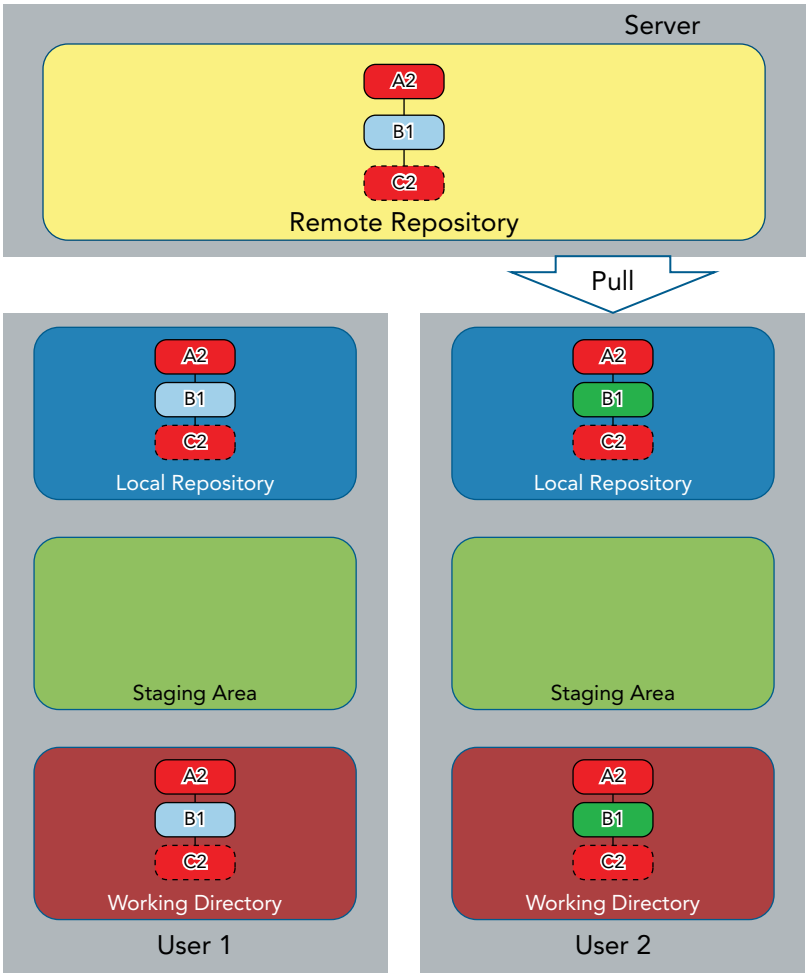


FIGURE 13-6: User 2 pulls the latest changes to merge updates locally.

Both users have cloned the repository down with the initial version of the file. User 1 makes two changes, stages and commits them, and then pushes them.

```
$ echo "Chapter 1" >> instructions.txt

$ git commit -am "Add chapter 1 heading"
[master 6d30ad0] Add chapter 1 heading
 1 file changed, 1 insertion(+)

$ echo "Welcome" >> instructions.txt
```

```
$ git commit -am "Add chapter 1 title"
[master ae619aa] Add chapter 1 title
 1 file changed, 1 insertion(+)

$ git push
Counting objects: 6, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (6/6), 546 bytes | 0 bytes/s, done.
Total 6 (delta 2), reused 0 (delta 0)
To C:/Program Files/Git/./calc2.git
   b7f554d..ae619aa  master -> master
```

The log now looks like this:

```
$ git log --oneline

ae619aa Add chapter 1 title
6d30ad0 Add chapter 1 heading
b7f554d Add initial instructions file
```

User 2 also makes two changes, then stages and commits them.

```
$ echo "Chapter 2" >> instructions.txt

$ git commit -am "Add chapter 2 heading"
[master 5446cab] Add chapter 2 heading
 1 file changed, 1 insertion(+)

$ echo "Next steps" >> instructions.txt

$ git commit -am "Add chapter 2 title"
[master c6495d9] Add chapter 2 title
 1 file changed, 1 insertion(+)
```

User 2's log now looks like this:

```
$ git log --oneline
c6495d9 Add chapter 2 title
5446cab Add chapter 2 heading
b7f554d Add initial instructions file
```

User 2 then attempts to push their changes over.

```
$ git push
To C:/Program Files/Git/./calc2.git
 ! [rejected]        master -> master (fetch first)
error: failed to push some refs to 'C:/Program Files/Git/./calc2.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

User 2 is rejected because User 1 got their changes in first. User 2 can now choose to use the default *git pull* merge behavior or the *git pull --rebase* option to do a rebase. Let's see what each operation looks like. First, I'll describe the default merge behavior of the basic pull operation.

Pull Alone

User 2 does a pull.

```
$ git pull
remote: Counting objects: 6, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 6 (delta 2), reused 0 (delta 0)
Unpacking objects: 100% (6/6), done.
From C:/Program Files/Git/./calc2
   b7f554d..ae619aa  master     -> origin/master
Auto-merging instructions.txt
CONFLICT (content): Merge conflict in instructions.txt
Automatic merge failed; fix conflicts and then commit the result.
```

As expected, there is a merge conflict.

```
$ cat instructions.txt
User Instructions
<<<<<<< HEAD
Chapter 2
Next steps
||||||| merged common ancestors
=======
Chapter 1
Welcome
>>>>>>> ae619aa1f2e09c7b98fefdaabf392765c43df61f
```

User 2 fixes the merge,

```
$ cat instructions.txt
User Instructions
Chapter 1
Welcome
Chapter 2
Next steps
```

then stages and commits it.

```
$ git add .

$ git commit -am "Merged chapter 2 content"
[master 934258f] Added chapter 2 content
```

After the commit, the log looks like this:

```
$ git log --oneline
934258f Merged chapter 2 content
c6495d9 Add chapter 2 title
5446cab Add chapter 2 heading
ae619aa Add chapter 1 title
6d30ad0 Add chapter 1 heading
b7f554d Add initial instructions file
```

Notice that you now have the new "merge commit" in your log. Now you'll see how this process differs when you use the rebase option.

Pull with the Rebase Option

Assume you've reset back to the point where User 2 has attempted to push and has received the rejection message. User 2's log looks like this:

```
$ git log --oneline
c6495d9 Add chapter 2 title
5446cab Add chapter 2 heading
b7f554d Add initial instructions file
```

User 2 now does the pull with the rebase option.

```
$ git pull --rebase
From C:/Program Files/Git/./calc2
 + 934258f...ae619aa master     -> origin/master  (forced update)
First, rewinding head to replay your work on top of it...
Applying: Add chapter 2 heading
Using index info to reconstruct a base tree...
M       instructions.txt
Falling back to patching base and 3-way merge...
Auto-merging instructions.txt
CONFLICT (content): Merge conflict in instructions.txt
error: Failed to merge in the changes.
Patch failed at 0001 Add chapter 2 heading
The copy of the patch that failed is found in: .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".
```

Notice that you see the familiar rebase messages about replaying changes, and then the merge conflict as Git tries to do the first step of the rebase. (Note the 0001 reference in "Patch failed at 0001…", which refers to the first commit you're trying to apply.)

You are now in a rebasing state until you complete the merges needed for a successful rebase or abort. Let's take a look at the log at this intermediate point:

```
$ git log --oneline
ae619aa Add chapter 1 title
6d30ad0 Add chapter 1 heading
b7f554d Add initial instructions file
```

From the log, you can see that the process started with the content from the master branch and was at the step of trying to apply the first new commit on top of that content. Looking at the file with conflicts, you can see the details:

```
$ cat instructions.txt
User Instructions
<<<<<<< ae619aa1f2e09c7b98fefdaabf392765c43df61f
Chapter 1
Welcome
||||||| merged common ancestors
=======
Chapter 2
>>>>>>> Add chapter 2 heading
```

After fixing the merge conflict for this first change, User 2 stages it and then tells the rebase to continue.

```
$ git add .

$ git rebase --continue
Applying: Add chapter 2 heading
Applying: Add chapter 2 title
Using index info to reconstruct a base tree...
M       instructions.txt
Falling back to patching base and 3-way merge...
Auto-merging instructions.txt
CONFLICT (content): Merge conflict in instructions.txt
error: Failed to merge in the changes.
Patch failed at 0002 Add chapter 2 title
The copy of the patch that failed is found in: .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".
```

You have a merge conflict again, but notice that this is for the second change (0002) by User 2. Again, the contents of the file with the conflicts reflect this state:

```
$ cat instructions.txt
User Instructions
Chapter 1
Welcome
Chapter 2
<<<<<<< 5a81fc4b7a7d02947fb5de6c7af148fb921d484b

||||||| merged common ancestors
=======
Next steps
>>>>>>> Add chapter 2 title
```

User 2 now fixes this conflict.

```
$ cat instructions.txt
User Instructions
Chapter 1
Welcome
Chapter 2
Next steps
```

Still in the rebasing state, User 2 stages the change and then tells the rebase to continue.

```
$ git add .

$ git rebase --continue
Applying: Add chapter 2 title
```

After the rebase is complete, the log shows the commits in the order you expect. Note that unlike the default merge outcome, you do not have a new, separate merge commit in the log.

```
$ git log --oneline
7effec1 Add chapter 2 title
```

```
5a81fc4 Add chapter 2 heading
ae619aa Add chapter 1 title
6d30ad0 Add chapter 1 heading
b7f554d Add initial instructions file
```

So, either the merge or rebase option of the pull command can incorporate other people's changes. The merge option may be simpler because it only requires one merge in some cases. However, it also introduces a merge commit into the history. The rebase option can require more merging and care to resolve conflicts, but it also gives a cleaner history.

Pushing Updated Content

You now have the latest changes from the remote repository incorporated into your changes in the local environment. This makes the content in the remote repository an ancestor to your changes. You can now try again to push your changes back to the remote repository, as shown in Figure 13-7.

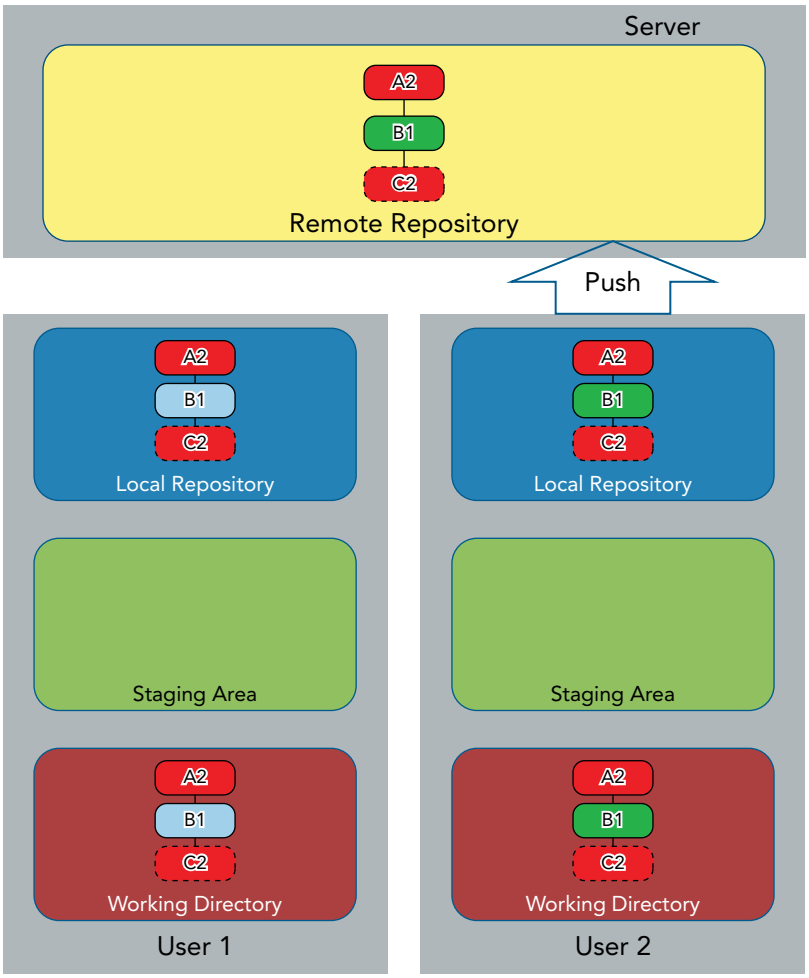


FIGURE 13-7: Merged content is pushed back into the remote.

The changes are merged successfully. This represents the typical model of working with Git and multiple users. If you try to push content and are rejected because of a non-fast-forward situation, you should pull or fetch the latest code, take care of any merge conflicts locally, and then try the push again. Of course, if someone else has made additional changes in the remote repository between when your push was initially rejected and when you try again, you may encounter new conflicts.

In the next section, I'll talk about an approach to how remote repositories are handled on many public hosting sites, and how it facilitates contributing to other projects.

HOSTED REPOSITORIES

As I discuss in the early chapters of this book, there are multiple public websites that can host Git repositories. There are also multiple packages that allow companies and groups to set up their own private, internal sites to host Git repositories. The repositories on these sites are the remote repositories, and the URL of the hosting site becomes the main part of the URL for the remote repository that's hosted there. You establish an account on the site, set up any necessary credentials, push your repository over to the site, and then you can treat it just like any other remote for cloning, fetching, pulling, and so on.

The public sites usually host repositories that are also public (viewable and clonable by anyone) for free. If you want to limit visibility and access to your repository, there is a cost associated with that.

On top of the basic hosting, these sites or packages layer on other features. Most of these features fall into two categories: support for working with your content and support for collaborating with others.

Some examples from the first category, support for working with your content, may include the following:

- Guidance for structuring your repositories, such as recommendations for adding README files
- Simplified interfaces to do operations with your repository, such as creating branches or showing differences
- Context-sensitive viewers to browse code, and even simple editors to let you create supporting content for your repository
- Ability to create and host releases of deliverables built from your content

Examples from the second category, support for collaborating with others, can include the following:

- Ability to manage group access by creating teams, groups, and so on
- Advanced collaboration tools, such as code-review facilities
- Ability to contribute changes to others' projects in a controlled manner through workflows such as *fork and pull*

This last item deserves further explanation because it is the foundation for the way many open-source projects are now managed. I'll now explore this model in more detail.

Models for Collaboration with Git

Over the years, several models have been developed to allow multiple people to work on the same project and code base in a Git repository. The simplest scenario is that everyone clones from the same remote repository, makes their changes, and pushes them back to the same repository. This can work well for a small number of users, but it doesn't help guarantee the readiness or the quality of the code that is pushed. It also promotes a *whoever gets there first* mentality to get pushes into the repository ahead of others. There is no controlled flow.

For projects with a few developers, this can be manageable but also inconvenient. For projects with a large number of developers, such a free-for-all can quickly become unmanageable and result in more time spent sorting out merge issues and other problems than actually developing content. The fork-and-pull model addresses some of these issues.

The Fork-and-Pull Model

In the fork-and-pull model, each developer or contributor has their own space where their remote repositories exist independent of the remote repositories of other users. If a user wants to just update their own projects, they can work with their remote repositories as usual.

However, if a user wants to contribute to another project (as is the case with most open-source projects), then they utilize a different workflow. To start with, the user who wants to contribute to the project owned by another user *forks* the owner's remote repository on the site. In this case, *fork* means getting a copy of the owner's repository (as it is at that point in time) and putting it in the contributor's own space. So, the contributor ends up with their own personal copy of the repository to use for developing their potential contributions. Figure 13-8 illustrates this process.

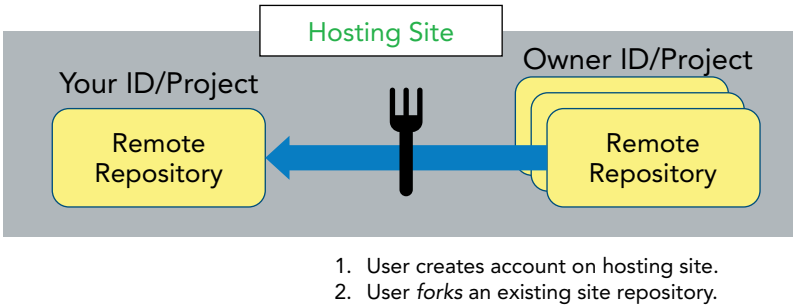


FIGURE 13-8: Forking a repository

They can then use the standard clone, add, commit, push, or other workflow to create a local environment and make changes to their copy of the owner's remote repository.

Sound familiar? In some sense, you can think of this as cloning a remote repository from one user to another. The result is that the contributor can then develop whatever content they want without

disturbing or interfering with the owner's repository. (For ease of later incorporation and changes, it is a good idea to make these changes on a topic branch.) Figure 13-9 illustrates this part of the process.

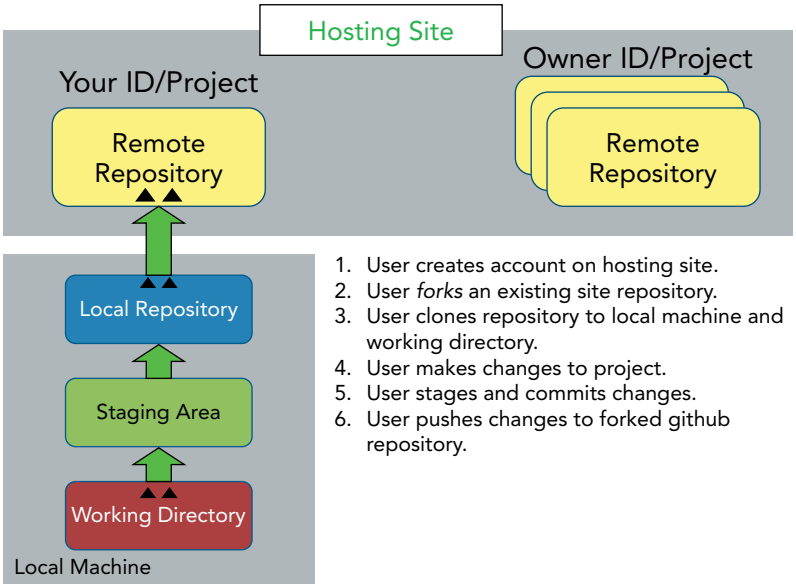


FIGURE 13-9: The typical Git lifecycle on a forked repository

So, once the contributor has completed their development and made the changes they think would be useful to the owner's project, how do they incorporate their updates into the owner's project? This is where the controlled process comes in. When the contributor is ready, they can send a request to the owner to merge their change in. This is called a *pull request* and can be as simple as a personal e-mail (although some sites provide an interface to create and send the request from directly within the browser, as shown in Figure 13-10.)

The owner can then review the candidate change from the contributor's repository and review it before agreeing to incorporate it. They can communicate with the contributor to ask questions, request changes, or offer feedback. If the owner decides that the change is a good addition to the overall project, and is of the quality that they expect, they can merge the change from the contributor into the primary repository for the project, as shown in Figure 13-11. (Depending on the site or package, there may be built-in mechanisms to do this on the site, or it may be necessary to clone, merge, and push.) However, if the change isn't appropriate or doesn't meet their standards, the owner can decline it or request changes to make it suitable for incorporation.

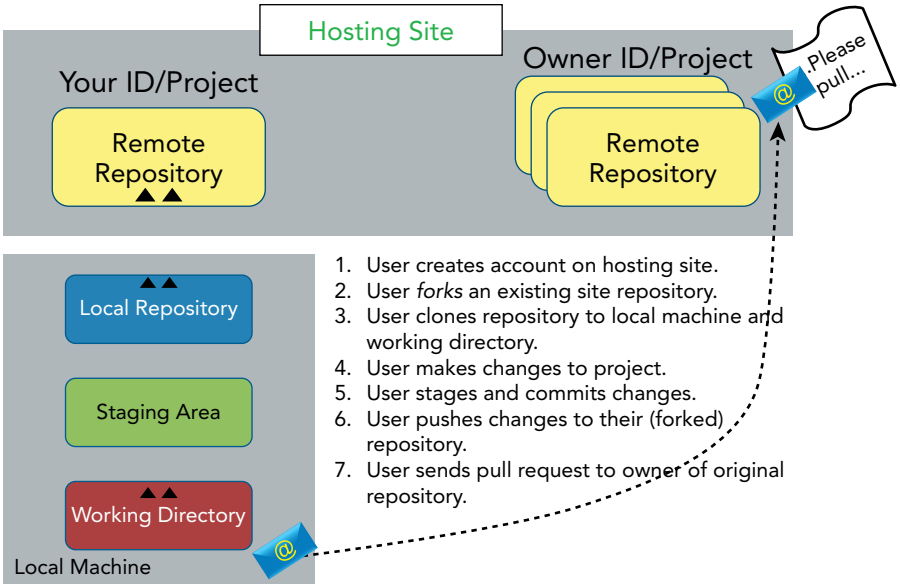


FIGURE 13-10: Sending a pull request to the owner

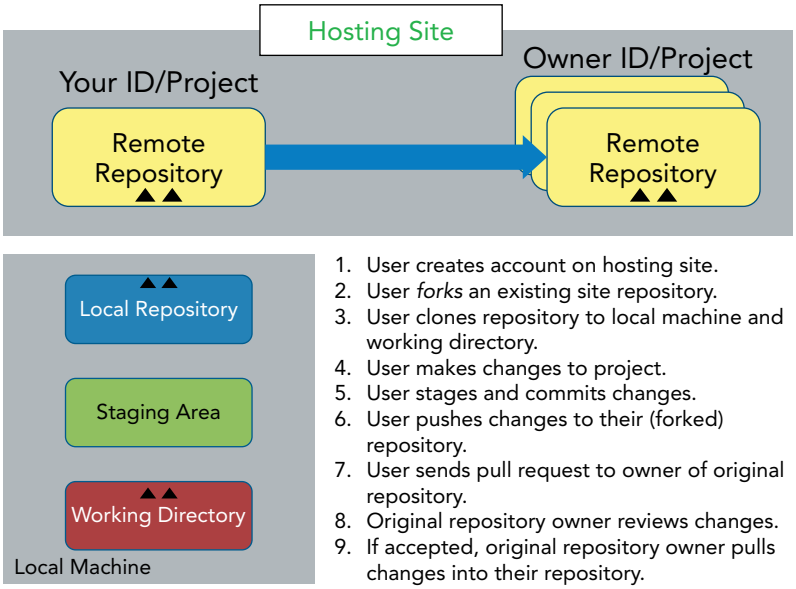


FIGURE 13-11: Repository owner pulls changes.

The *fork* operation to create a copy of the owner's remote repository in the contributor's space, coupled with the request to *pull* the change and incorporate it, give the *fork-and-pull* model its name. This model provides the project owner with a high degree of oversight (and responsibility) because they are able to review and decide whether or not to take changes from others. The criteria for accepting a change could be anything from a visual inspection to a group code review to requiring passage of a set of automated testing. The end benefit is that the primary project repository (that of the owner) does not have any changes merged into it except those that the owner allows and that pass any quality gates that may be set up (such as having unit tests, adhering to a coding style, and so on).

In some cases, applications may be so big or have so many contributors that even the fork-and-pull approach becomes difficult to manage. In that scenario, one strategy is to break the application up into multiple projects (assuming it isn't already) and assign different owners to oversee contributions to each of the parts. This is also sometimes done in a hierarchical fashion where owners of lower-level components oversee updates to those components and then submit pull requests to owners of higher-level sections to pull their collected changes in, and so on. This is similar to using military ranks for the different levels of owners—sergeants, lieutenants, generals, and so on. You may sometimes see these terms used to describe the hierarchy of owners for different parts of large projects.

Use of Multiple Remotes

When someone is working on a contribution for another project, there is a useful strategy that can help them keep their content in sync with the primary project, and guarantee that the owner of the project will be more likely to easily merge their changes. After forking the owner's project into their own space and cloning a local environment from it, the contributor has a remote reference that points back to the forked repository. However, during the period when the contributor is working on their changes, it's likely that other changes and contributions are being made to the owner's primary repository. If the contributor doesn't keep up with those changes in their forked repository, they can end up with a very different code base (aside from their changes) and significant merge issues when they are done. This can delay or prohibit their changes from being accepted. Put another way, the copy doesn't stay in sync with the original while new development is going on.

To avoid this problem, a simple strategy is to create a second remote reference to the original repository (the one the fork was created from) and periodically fetch updates from there. This is simple to do with the *git remote* command, which I discuss in Chapter 12. To use an illustration based on a GitHub repository, you may have forked a repository into your space and then cloned from it. As a result, you could have an *origin* like this:

```
$ git remote -v
origin  https://github.com/contributor/project1 (fetch)
origin  https://github.com/contributor/project1 (push)
```

To create a connection to the owner's project, you could create another remote reference like this:

```
$ git remote add primary https://github.com/owner/project1.git
```

You can then have access to both of these remotes:

```
$ git remote -v
origin  https://github.com/contributor/project1 (fetch)
origin  https://github.com/contributor/project1 (push)
primary https://github.com/owner/project1 (fetch)
primary https://github.com/owner/project1 (push)
```

The benefit of having this additional remote is that now, while you are working on your changes, you can periodically fetch the latest changes from the owner's repository.

```
$ git fetch primary <branch>
```

You can then use the *git merge* command to merge those latest changes in with your current work at your discretion. When you are ready to submit the pull request, if you've been diligent about fetching and merging the latest updates from the owner's repository, the owner should find it fairly easy to merge your changes—perhaps even doing a fast-forward. All of this contributes to the likelihood that your change will be incorporated quickly if it is suitable.

Having multiple remotes is an example of where you may want to frequently update your content in the local repository and the working directory, so that you can keep up to date with the changes being made in the remote of the other repository while you are working on your own changes. That way, you don't diverge too far from the original code base as it continues to be updated.

If the updates overlap your local changes, this can be problematic. You need to have a way to manage this process; I offer one possible strategy in the next section.

Managing Local Updates

In Chapter 12, I describe how the pull command fetches updates from the remote repository into the local repository and then attempts a merge. If the merge is successful, the working directory is updated as well as the local repository. In some cases, such as the one I previously described, automatically merging updates into the working directory may not always be what you want.

Suppose that you are implementing a new feature that involves changing files A, B, and C. You haven't completed your changes to A, B, and C when you learn that a key security fix has been made in the original code base that you need to incorporate in short order. So, you do a pull from the updated code base. The pull brings down the security fix and merges the changed files into the contents of your working directory. As it turns out, the security fix also involved changes to files A, B, and C. The merge changed the versions of A, B, and C that you were working on in a way that breaks your new feature significantly. You're now faced with trying to untangle what the automatic merge did after the fact to get back to a working state.

A better approach is to have control over the merge before it occurs. Using my example, that approach would look like this:

1. Save off your changes that are in progress in A, B, and C before you do the pull. You do this using the *git stash* command that I cover in Chapter 11. Now you've stashed your changes that are in progress, effectively resetting your working directory and staging area back to the way they were at the last commit.
2. Do the pull operation to get the updates with the security fix into your local environment.
3. When you are ready, pop or apply your work from the stash. When you try to do this and there are differences between the stashed contents and the working directory contents, Git stops and informs you about the differences.
4. Make the necessary changes in your working directory, or even create another branch to do the changes in.

So, one model of incorporating updates from others while you have work in progress is to stash your changes, do a pull, and then apply or pop the changes from the stash. At that point, Git lets you know if there are merge conflicts, and you can resolve them manually instead of trying to unravel what the pull may have changed automatically.

Putting It All Together

I've now covered several different aspects of working with changes and remotes for updates and merge situations. Although I've presented these situations in the context of working with multiple users, it is certainly possible to have similar merge or update situations with a single user working in multiple branches and pushing one set of changes from a branch to the remote repository before another set of changes.

Figure 13-12 ties together the different areas I've been talking about into a single representation of the workflow for dealing with changes and remotes. The left half of the figure focuses on updating the local environment, while the right half focuses on updating the remote repository. This is certainly not the only workflow that you can use, but it will give you an idea of how all the parts I have talked about can be used together.

Keep in mind that, although I don't show it in the figure, you might have multiple remotes that you choose to update from, and your initial remote for cloning might have been forked from another one if you are using a hosting site or package that supports that functionality.

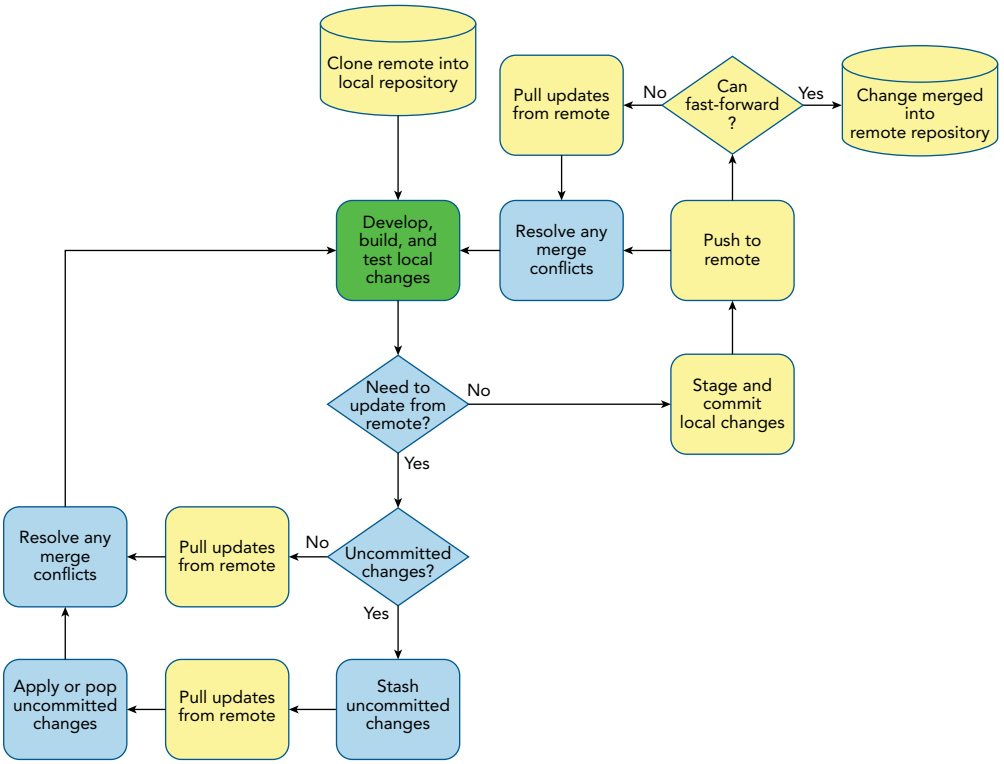


FIGURE 13-12: A workflow model for making and incorporating changes

SUMMARY

In this chapter, I've explained how to work in a Git environment with multiple users. The key is to understand that a merge conflict in Git occurs when any update has been made in the same commit that you are trying to pull or push since you last retrieved a copy from the remote repository. In Git, the scope of change is at the commit (snapshot) level, not the individual file level.

I also covered a common model used by many hosting sites and packages to allow collaboration and contribution back to projects: the fork-and-pull model. This model involves copying another remote repository into your own space (forking), developing against that copy, and then requesting the owner of the remote repository that you forked to *pull* in your changes if they approve. I also showed that a useful construct for keeping up with changes in the original repository is to create a secondary remote reference and fetch changes periodically from that reference into your forked environment.

I described a common strategy to insulate yourself from potentially challenging automatic merge situations when you have work in progress and need to pull from the remote repository. You can use the Git stash functionality to save your uncommitted changes, do a clean pull, and then apply or pop from the stash to better control merging your work in progress back into the latest code from the remote side.

Finally, I presented a workflow diagram that provides one example of tying these key concepts and strategies together for working with remotes and updates from other users.

In the next chapter, I explore a new way that Git allows you to work in multiple branches concurrently. I also look at constructs that Git provides to allow for treating multiple repositories and projects as a unit.

About Connected Lab 9: Using the Overall Workflow with a Remote Repository

In lab 9, you'll get to see what it's like when multiple users are making changes to the same repository. This is simulated through working with multiple local clones and pushing back to the same remote.

This lab also provides additional experience with merging and rebasing as part of this simulation. This lab is longer than most, but it is a valuable exercise to complete as you will encounter situations like this routinely when working with Git.

Connected Lab 9

Using the Overall Workflow with a Remote Repository

In this lab, you'll simulate working in an environment where multiple users are making changes to a remote repository. You'll see how to deal with getting rejected by Git, and also get some practice with techniques like rebasing.

PREREQUISITES

This lab requires that you have Internet access and have completed Connected Lab 8: Setting up a GitHub Account and Cloning a Repository. You will be working from the same directory that you used in Connected Lab 8.

STEPS

1. At the end of Connected Lab 8, you cloned a repository from GitHub into two different directories on your local system: calc2 and calc_other. You'll use these two directories to simulate two users working against the same remote repository. Change into the calc_other directory.

   ```
   $ cd calc_other
   ```

2. To see what features your calculator already has, open up the calc.html program in a browser and take a look. Note that you have access to the basic arithmetic functions: addition, subtraction, multiplication, and division.

3. You want to incorporate some other features into your calculator program from the features branch. First, you need to set up a local features branch, so create a local branch that tracks the remote features branch.

   ```
   $ git branch features origin/features
   ```

4. Look at what features are available for you to use.

```
$ git log --oneline features
```

5. Now, let's create a temporary branch to use as we work on incorporating these features. Create a new branch cpick and switch to it in one step using the shortcut. (cpick here represents "cherry pick")

```
$ git checkout -b cpick
```

6. You want to pull in the max, exp, and min functions to add to your calculator. Start with the max function. First, find the SHA1 value of the max function from the history listing in step 4 of the features branch.

7. Issue the command to cherry-pick that commit's SHA1 value (the one from step 4) onto your current branch.

```
$ git cherry-pick d003b91
```

8. A message appears that the function is being added. Assuming there are no errors, run a log of your current branch.

```
$ git log cpick --oneline
```

Notice that the log now shows the max function has been incorporated into the branch.

8. Open up calc.html in the browser (or refresh it if you already have it open) to verify you now have the new max function (and no other new functions). To verify this, click the drop-down menu, between the two number input fields, and make sure that max shows up as one of the available operations.

9. Now, you can use rebase to incorporate the exp and min functions. This way, you will have the functionality incorporated, as well as the history.

10. From the log, find the SHA1 value for the exp function. Use the following command to execute the desired rebase:

```
$ git rebase 3753e5a
```

11. Once the rebase is executed, you can do a quick log of your current branch (cpick) and verify that the history records now show up there. You can also open the calc.html program in a browser to verify that the functions are there. (The exp function will be represented as "***".)

```
$ git log --oneline
$ <start|open> calc.html
```

12. Now that you have your code as desired in the cpick branch, you're ready to merge that branch back into the master branch. Run the following commands:

```
$ git checkout master
$ git merge cpick
```

13. Once the merge is complete, push the changes out to the remote.

```
$ git push
```

14. You are done simulating the activity of user one. Now, you can move on to working as user two in the calc2 area. Change to the calc2 subdirectory.

```
$ cd ../calc2
```

15. User two wants to merge the ui work from origin/ui into the master branch to promote it. For convenience, you first need to create a local ui branch from the remote branch.

```
$ git branch ui origin/ui
```

16. Merge the ui branch into master. (If you are not still on master, check out master first.)

```
$ git merge ui
```

17. Assuming the merge is successful, push the updates out to the remote.

```
$ git push origin master
```

18. Your push is rejected. This is because the push done by the other user (as simulated when you were working in calc_other) changed the same commit and was pushed before you pushed your updates. Git determines that it can't do a fast-forward merge and so rejects the push.

19. It is now up to you to resolve this push issue. You could do a force push (with the -f option), but that is often dangerous. Instead, try the suggestion that Git offers and do a pull to see if it can merge cleanly.

```
$ git pull origin master
```

20. Notice again that you have a merge conflict. You could certainly go into the file and edit it to resolve the various merge conflicts. Or, you could use one of the merge strategies to force choosing one version or the other. However, in this case, you decide to incorporate the history as well. That points you toward another option: a rebase. Before you can try that, you need to abort this merge.

```
$ git merge --abort
```

21. Now you can get the updated content without having Git try to merge it locally. Recall that the fetch command updates the remote tracking branches but not the local branches. So, you execute a fetch command.

```
$ git fetch
```

22. You are now ready to try the rebase. If all goes well, this operation will rebase locally off of the updated content from the master in the remote tracking branches. Run the following command (making sure you are in the master branch when you do so):

```
$ git rebase origin/master
```

23. Once again, conflicts arise. Because the changes are different, between adding functionality and changing ui features, you can make an educated guess that if you just keep the current changes and apply the other changes on top of them, you won't run into critical conflicts. So, you can tell Git to keep your changes if there are perceived conflicts. The easiest way to do that is to add the -Xours option. Recall that this option passes the "keep ours if

there's a conflict" option to the default recursive strategy. Abort the current rebase operation, and run the command again with the extra option.

```
$ git rebase --abort
$ git rebase -Xours origin/master
```

24. This time it works. (If it didn't work, you could have aborted the rebase again.) Do a quick git log to see if the commits look correct.

```
$ git log --oneline
```

25. Note that all of the commits that you would expect are now there. Try pushing these updates over to the remote side again.

```
$ git push
```

26. This time, it pushes with no problems.

14

Working with Trees and Modules in Git

WHAT'S IN THIS CHAPTER?

- Working with changes in multiple branches simultaneously using worktrees
- Including repositories as subprojects with submodules
- Coordinating changes in submodules with the containing project
- Addressing potential problems when using submodules
- Incorporating projects as subdirectories with subtrees

In this book, you've primarily been working with single projects managed in a single repository where you worked on only one branch at a time. This works well for most projects, but there are times when you need to extend this model. Two such examples include working on multiple branches concurrently in a project, and including other repositories as subprojects or subdirectories.

WORKTREES

As I discuss in Chapter 8, one nice feature of Git is that you can use the same working directory for all of the branches you need to work with. However, as it turns out, this can also be a liability.

In the past, if you were making changes in one branch and needed to switch to a new branch, you had three choices: commit your changes to get to a clean working directory, stash your changes that were in progress, or create a separate clone of the repository in a different area and work on the other branch there.

Starting with version 2.5, Git formally introduced a more workable alternative: *worktrees* (working trees). The idea with worktrees is that you can have multiple, separate working directories, all connected to the same staging area and local repository. The traditional working directory that I've been using throughout this book is called, in Git terminology, the *main working tree*, while any new trees you create with this command are called *linked working trees*.

To use separate working trees, Git introduced a new *worktree* command. The syntax is shown here:

```
git worktree add [-f] [--detach] [-b <new-branch>] <path> [<branch>]

git worktree list [--porcelain]
git worktree prune [-n] [-v] [--expire <expire>]
```

Notice that the worktree command has three subcommands: *add*, *list*, and *prune*. Any option must be preceded by one of the subcommands. In the following sections, I'll briefly cover each subcommand.

Adding a Worktree

The first worktree subcommand (add) is designed to add a new worktree for working with a particular branch. Its simple syntax,

```
$ git worktree add <path> [<branch>]
```

creates a new working directory in the <*path*> location with a checked-out copy of <*branch*>. For example, if you have a project that has a *docs* branch and you want to work with that branch in a separate directory named *tmparea*, you can use the following command:

```
$ git worktree add ../tmparea docs
Preparing ../tmparea (identifier tmparea)
HEAD is now at a83878d add info on button
```

The last line here indicates the most recent commit on the docs branch.

If you now switch to the new area, you see by the prompt that you have a checked-out copy of the docs branch, just as if you had cloned a new copy of the repository and changed to the branch.

```
$ cd ../tmparea
 ~/tmparea (docs)
```

What if you want to work on yet another copy of the docs branch? Attempting to add another area with the docs branch results in an error message:

```
$ git worktree add ../tmparea2 docs
fatal: 'docs' is already checked out at 'C:/Users/bcl/tmparea'
```

This is a general safeguard. If you want to work around it, you can add the --force option, as shown here:

```
$ git worktree add --force ../tmparea2 docs
Preparing ../tmparea2 (identifier tmparea2)
HEAD is now at a83878d add info on button
```

You can also create a new worktree with a different branch name based on an existing branch. To do this, you pass the -b or -B option with the desired new branch name.

```
$ git worktree add -b fixdocs ../tmparea3 docs
Preparing ../tmparea3 (identifier tmparea3)
HEAD is now at a83878d add info on button
```

This command tells Git to create a new branch named *fixdocs* off of the existing *docs* branch in the ../tmparea3 subdirectory.

By default, the worktree command doesn't let you create a new branch that has the same name as an existing branch. The -B option allows you to force having a new branch with the same name as an existing branch.

```
$ git worktree add -b docs2 ../tmparea4 docs
fatal: A branch named 'docs2' already exists.

$ git worktree add -B docs2 ../tmparea4 docs
Preparing ../tmparea4 (identifier tmparea4)
HEAD is now at a83878d add info on button
```

What happens if you don't supply a branch name to create? The worktree command creates a new branch with the same name as the target area and based on whatever branch is current in the main working tree.

```
$ git branch
  cpick
  docs
  docs2
  features2
  fixdocs
* master
  tmpdocs

$ git log -1 --oneline
06efa5e update field size

$ git worktree add ../tmparea5
Preparing ../tmparea5 (identifier tmparea5)
HEAD is now at 06efa5e update field size

$ cd ../tmparea5

~/tmparea5 (tmparea5)
$ git log -1 --oneline
06efa5e update field size
```

Finally, if you want to work in a detached mode (for example, to later create your own branch name on the area), you can use the --detach option.

```
$ git worktree add --detach ../tmparea6
Preparing ../tmparea6 (identifier tmparea6)
HEAD is now at 06efa5e update field size
```

```
$ cd ../tmparea6

~/tmparea6 ((06efa5e...))
$ git status
Not currently on any branch.
nothing to commit, working directory clean
```

> **NOTE** *Git stores the information about working trees in the Git directory area. Assuming the Git directory maps to .git, the working tree information is stored in .git/worktrees/*<name of worktree>.
>
> *Taking a closer look at one of these worktree areas, you can see some of the information you would typically expect in a working directory area.*
>
> ```
> $ ls .git/worktrees/tmparea
> commondir  gitdir  HEAD  index  ORIG_HEAD
> ```
>
> *In this case, gitdir is set to point to this worktree's .git directory, and commondir is set to point back to the main worktree's area.*

Figure 14-1 illustrates a working tree setup.

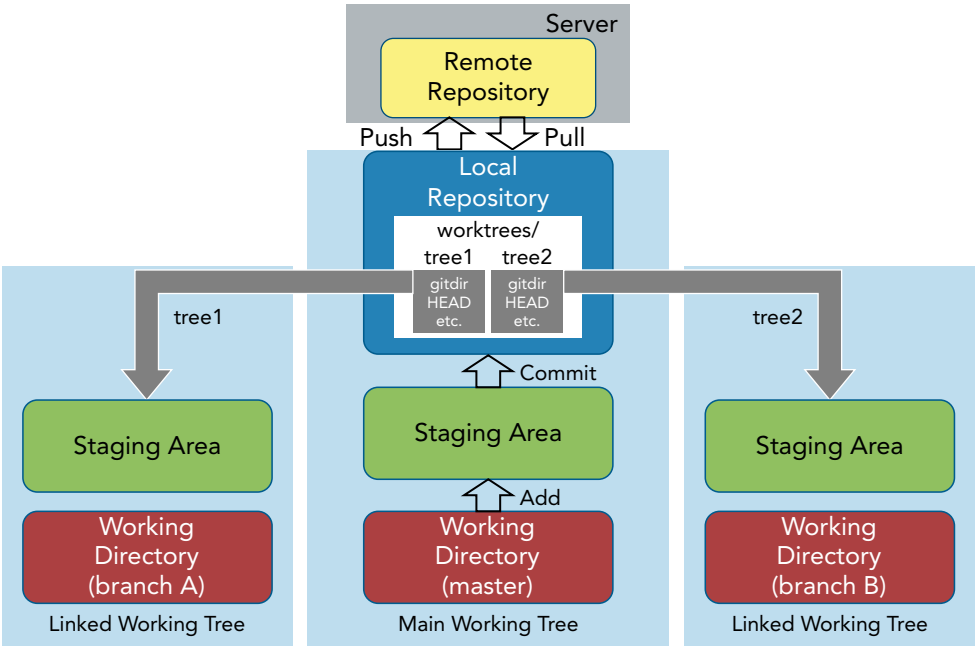


FIGURE 14-1: Illustration of multiple working trees

Listing Out the Working Trees

The second subcommand for worktree is *list*. As the name implies, this subcommand allows you to list out details about the set of working trees that are currently active for this repository.

Using the list subcommand is straightforward.

```
$ git worktree list
C:/Users/bcl/calc2     06efa5e [master]
C:/Users/bcl/tmparea   a83878d [docs]
C:/Users/bcl/tmparea2  a83878d [docs]
C:/Users/bcl/tmparea3  a83878d [fixdocs]
C:/Users/bcl/tmparea4  a83878d [docs2]
C:/Users/bcl/tmparea5  06efa5e [tmparea5]
C:/Users/bcl/tmparea6  06efa5e (detached HEAD)
```

There is only one option for list: *porcelain*. This option lists the worktree information in a more verbose format that may be easier for scripts to process and that should be consistent across future versions of Git.

```
$ git worktree list --porcelain
worktree C:/Users/bcl/calc2
HEAD 06efa5ecedc5db8b4834ffc0023facb70053d46e
branch refs/heads/master

[other branches...]

worktree C:/Users/bcl/tmparea6
HEAD 06efa5ecedc5db8b4834ffc0023facb70053d46e
detached
```

Pruning a Worktree

Finally, there is the prune subcommand. As its name implies, the prune subcommand removes worktree information. However, it only removes the information from the Git directory (.git) *after* the actual worktree subdirectory has been manually removed. Here is an example from the main worktree:

```
$ rm -rf ../tmparea6
$ git worktree prune
```

The prune subcommand has two simple options:

- -n (--dry-run)—This option tells Git to not execute, but to just explain what it would do.

  ```
  $ rm -rf ../tmparea4
  $ git worktree prune -n
  Removing worktrees/tmparea4: gitdir file points to non-existent location
  ```

- -v (--verbose)—This option tells Git to be more verbose in explaining what it's doing.

  ```
  $ git worktree prune -v
  Removing worktrees/tmparea3: gitdir file points to non-existent location
  Removing worktrees/tmparea4: gitdir file points to non-existent location
  ```

Notice that in the verbose operation, I also pruned tmparea4, because I had run the prune subcommand on that area with the dry-run option without actually executing the prune operation.

> **NOTE** *If you create a working tree on removable media, you may want to have it persist even when the media is not mounted and a prune is done. To do this, you add a file named* locked *in the directory for the worktree under .git/worktrees/*<worktree name>*. For example, if you create a working tree named* brentsfix *on your removable media, then to have this tree persist, you can create .git/worktrees/brentsfix/locked. The convention is for the text of the* locked *file to have the reason the area needs to be persisted (in plain text) inside of it.*

SUBMODULES

Sometimes you may need to include a separate repository along with your current repository. The most common use case for this would be to include the Git repository for one or more dependencies along with the original repository for a project. Git offers a way to do this through functionality called *submodules*. This means that you have a subdirectory off of your original repository that contains a clone of another Git repository. Your original repository stores metadata in its Git directory about the existence of the subdirectory and repository and what the current commit is in the clone. Another name for this original repository is the *superproject* as used in the Git documentation. I'll use that name as well.

You can treat the repository in the subdirectory (submodule) independently like any other repository. However, if you update something in a submodule, you have to perform extra steps to update which commit in the submodule the superproject points to. Otherwise, things can get confusing and badly out of sync.

Traditionally, submodules have received a bad reputation because of their limitations, which make it easier to get into difficult states. However, they do have some valid uses. The syntax for the submodule command in Git is as follows:

```
git submodule [--quiet] add [-b <branch>] [-f|--force] [--name <name>]
              [--reference <repository>] [--depth <depth>] [--] <repository> [<path>]
git submodule [--quiet] status [--cached] [--recursive] [--] [<path>...]
git submodule [--quiet] init [--] [<path>...]
git submodule [--quiet] deinit [-f|--force] [--] <path>...
git submodule [--quiet] update [--init] [--remote] [-N|--no-fetch]
              [-f|--force] [--rebase|--merge] [--reference <repository>]
              [--depth <depth>] [--recursive] [--] [<path>...]
git submodule [--quiet] summary [--cached|--files] [(-n|--summary-limit) <n>]
              [commit] [--] [<path>...]
git submodule [--quiet] foreach [--recursive] <command>
git submodule [--quiet] sync [--recursive] [--] [<path>...]
```

As I said earlier, submodules allow you to have a separate repository as a subdirectory in your superproject. Because the repository is separate, it still maintains its own history. And the submodule is *not* automatically updated by default when the superproject is updated via one of the interactions with remotes (which I discuss in Chapter 12). You can do more direct management of these submodules with the options of the submodule command.

To track the submodule information, Git creates and manages a *.gitmodules* file at the root of the repository. This file contains entries in the following format:

```
[submodule <name>]
        path = <relative path>
        url = <url for cloning, updating, etc.>
```

It's worth clarifying here that submodules are not the same as remotes. Remotes are server-side or public copies of the same repository, while submodules are just other repositories that you want to use or include as dependencies—but as different repositories. Figure 14-2 illustrates a submodule arrangement.

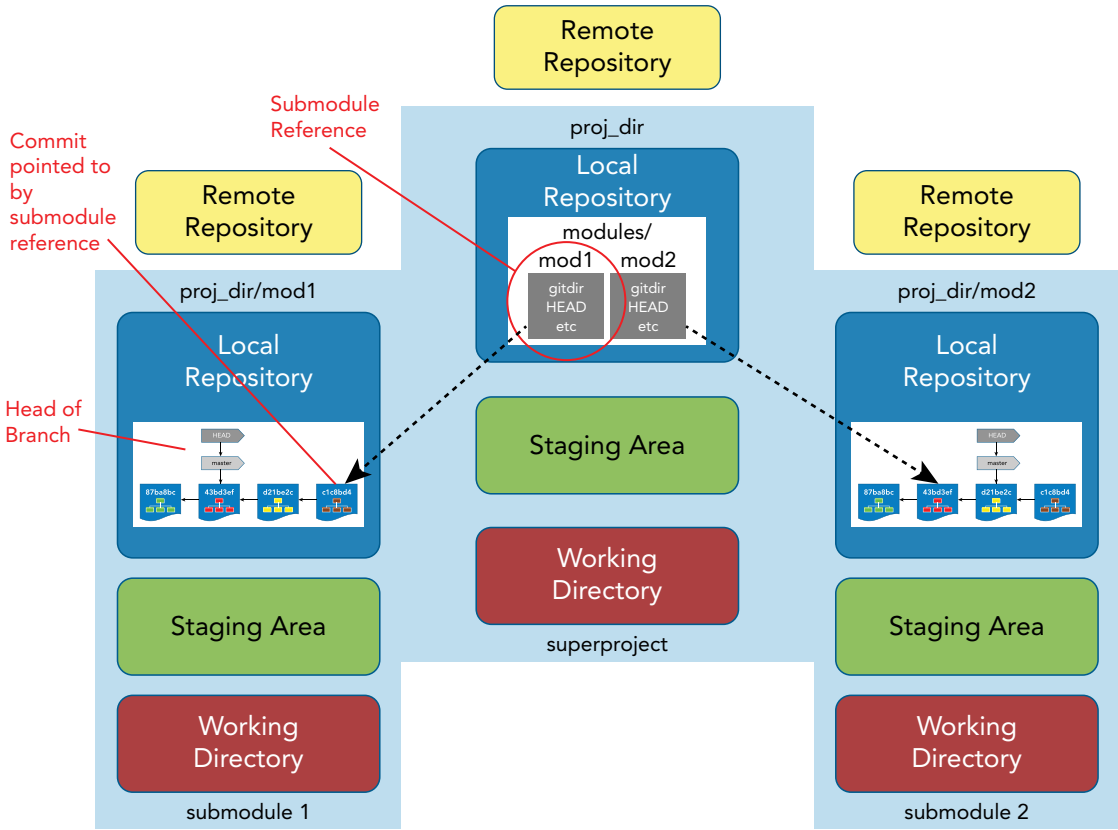


FIGURE 14-2: Illustration of how submodules work

Understanding How Submodules Work

To understand how submodules work, you'll look at them from two perspectives:

1. Creating a new set—The perspective of the user who adds submodules to a superproject and pushes the set to a remote.
2. Cloning an existing set—The perspective of a user who clones down a copy of the superproject with the submodules from the remote.

To associate a new set of submodules to an existing project, you use the submodule *add* command.

Adding a Submodule

In the following example, two submodules are added to an existing project (repository). This project will be the superproject. Assuming you've already created and pushed projects named *mod1* and *mod2*, you can add the submodules to the superproject with the following commands:

```
$ git submodule add <url to mod1> mod1
$ git submodule add <url to mod2> mod2
```

The submodule command's add operation does several things as indicated by the following steps. The output after each step shows the results.

1. Git clones down the repository for the submodule into the current directory.

```
$ git submodule add <remote path for mod1> mod1
Cloning into 'mod1'...
done.

$ git submodule add <remote path for mod2> mod2
Cloning into 'mod2'...
done.
```

2. By default, Git checks out the master branch.

```
$ cd mod1
$ git branch
* master


$ cd ../mod2
$ git branch
* master
```

3. Git adds the submodule's path for cloning to the .gitmodules file.

```
$ cat .gitmodules
[submodule "mod1"]
        path = mod1
        url = <remote path for mod1>
[submodule "mod2"]
        path = mod2
        url = <remote path for mod2>
```

4. Git adds the .gitmodules file to the index, ready to be committed.
5. Git adds the current commit ID of the submodule to the index, ready to be committed.

```
$ git status
On branch master
  ...
        new file:   .gitmodules
        new file:   mod1
        new file:   mod2
```

Once the submodules' paths are recorded in the .gitmodules file, they are linked there to be included with any future cloning of the main project.

To finish the add process for mod1 and mod2, you just need to complete the Git workflow by committing and pushing the submodule-related changes that the add command has staged for you. You do this from the superproject's directory.

```
$ git commit -m "Add submodules mod1 and mod2"
[master 2745a27] Add submodules mod1 and mod2
 3 files changed, 8 insertions(+)
 create mode 100644 .gitmodules
 create mode 160000 mod1
 create mode 160000 mod2

$ git push
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 400 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To C:/Users/bcl/submod/remote/main.git
   941450a..d116ad1  master -> master
```

Here, you have told Git to associate two other repositories with your repository as connected. Git manages this, in part, by creating the .gitmodules file to map which subdirectories the submodule content should go into, and storing information for each module that contains the name of the module and the SHA1 value of the current commit in the module. This information is then pushed to the remote side so the connection information is stored with the project when it is cloned in the future.

So, you're pushing mapping information to the remote repository for your superproject that tells it how to find, map, and populate the submodules that you want to use. However, note that you are not pushing any changes to the repository for the submodules themselves. I will talk about this later in this chapter.

At this point, let's use another git submodule command to see the status of your changes.

Determining Submodule Status

As the name implies, this submodule subcommand is used to see the status of the various submodules associated with a project. In particular, this command shows the SHA1 value for the (currently

checked-out) commit for a submodule, with the path. The output also includes a simple prefix character, defined as:

"-" if the submodule is not initialized

"+" if the submodule's current version that's checked out is different from the SHA1 in the containing repository

"U" if there are merge conflicts in the submodule

If you look at the current status of the submodules you just added, you see something like this:

```
$ git submodule status
 8add7dab652c856b65770bca867db2bbb39c0d00 mod1 (heads/master)
 7c2584f768973e61e8a725877dc317f7d2f74f37 mod2 (heads/master)
```

As noted earlier, the first field contains the SHA1 values for the currently checked-out commits in each of the submodules. This is followed by the local name you assigned to the submodule when you added it to your project. To further demonstrate this mapping, you can go into the submodule area, and do a quick log:

```
$ cd mod1
$ git log --oneline
8add7da Add initial content for module mod1.
```

Note that the SHA1 value of the current (only) commit there matches the SHA1 value shown in the output of the earlier status command.

NOTE *Internally, Git stores module information for submodules in a directory named .git/modules. Inside this area, there is a separate subdirectory for each of the modules attached to this project. In the example you've been working with, if you were to go into this area, you would see something like this:*

```
$ ls .git/modules/*
.git/modules/mod1:
config       HEAD    index  logs/     packed-refs
description  hooks/  info/  objects/  refs/

.git/modules/mod2:
config       HEAD    index  logs/     packed-refs
description  hooks/  info/  objects/  refs/
```

Cloning with Submodules

Now, let's switch to the perspective of another user who wants to clone down and work with the project with its submodules. First, you create a separate area, and clone your project with the submodules into it.

```
$ git clone <remote path>/main.git
Cloning into 'main'...
done.
```

```
$ cd main

$ ls -a
./  ../  .git/  .gitmodules  file1.txt  mod1/  mod2/
```

So, it appears you cloned down the superproject and the submodules. However, take a look at what's in the submodule directories:

```
$ ls mod1
$ ls mod2
```

Nothing shows up there—why? Let's use the submodule status command to see what the status is of the submodules.

```
$ git submodule status
-8add7dab652c856b65770bca867db2bbb39c0d00 mod1
-7c2584f768973e61e8a725877dc317f7d2f74f37 mod2
```

Notice the dash sign (-) in the first column. As previously mentioned in the section on the status command, the dash sign means the submodules have not been initialized.

In this instance, not being initialized equates to your superproject not knowing about the modules. The directories for the submodules exist but haven't been populated. More importantly, information about the submodule locations (from the .gitmodules file) hasn't been put into the superproject's config file yet. This is what the submodule init command will do for you.

```
$ git submodule init
Submodule 'mod1' (<remote path>/mod1.git) registered for path 'mod1'
Submodule 'mod2' (<remote path>/mod2.git) registered for path 'mod2'
```

After running this command, you have the remote information in your config file for the repository.

```
$ git config -l | grep submodule
submodule.mod1.url=<remote path>/mod1.git
submodule.mod2.url=<remote path>/mod2.git
```

This completes the *init* step. However, if you look into the repository directories after this, you'll notice that you still don't have any content. As it turns out, pulling down the submodules for an existing project with submodules is a two-step process.

The init subcommand registered the submodules in the superproject's configuration so it can reference them directly. Now you run the update subcommand for submodule to actually clone those repositories into your subdirectories and check out the indicated commits for the containing project.

```
$ git submodule update
Cloning into 'mod1'...
done.
Submodule path 'mod1': checked out '8add7dab652c856b65770bca867db2bbb39c0d00'
Cloning into 'mod2'...
done.
Submodule path 'mod2': checked out '7c2584f768973e61e8a725877dc317f7d2f74f37'
```

Why a two-step process? Having the init and update sub commands separated provides an opportunity for the user to update the URL (path) in the .gitmodules file if needed before cloning the submodule (that is, before the update command). If you don't need to do this, though, and you want to execute both operations with one command, there is a shortcut, as shown in the tip.

TIP *As you just saw, it is normally a two-step operation to populate your submodules: submodule init and submodule update. However, the update sub command has an --init option. You can use it as a shortcut to do the init and update subcommands with one call after the clone of the container project.*

```
$ git submodule update --init
```

Even more helpful, Git provides an option called --recursive that you can add to your clone command. Using this option includes the functionality of the submodule update --init, *simplifying things even further.*

```
$ git clone --recursive <URL of container project>
Cloning into 'main'...
done.
Submodule 'mod1' (<url to remote mod1>/mod1.git) registered for
path 'mod1'
Submodule 'mod2' ((<url to remote mod2>/mod2.git) registered for
path 'mod2'
Cloning into 'mod1'...
done.
Submodule path 'mod1': checked out
'8add7dab652c856b65770bca867db2bbb39c0d00'
Cloning into 'mod2'...
done.
Submodule path 'mod2': checked out
'7c2584f768973e61e8a725877dc317f7d2f74f37'
```

(Note that you can also use --recurse-submodules, which is the same as --recursive.)

You can use these options to save some steps whenever you need to clone a project that has submodules.

A key point to emphasize here is that this operation cloned the repositories for the submodules and checked out the commits that were current when the submodules were added.

If you go back into the separate, original repositories for mod1 and mod2, and do a log, you see that there have been some updates since you added these repositories as submodules.

```
$ cd <original separate mod1 path>/mod1; git log --oneline
a76a3fd update info file
8add7da Add initial content for module mod1.

$ cd <original separate mod2 path>/mod2; git log --oneline
cfa214d update 2 to info file
7c2584f update of info file
07f58e0 Add initial content for module mod2.
```

Now, if you look at the results of your submodule updates in the recently cloned repository, you see some differences.

```
$ cd mod1

mod1 ((8add7da...))
```

```
$ git log --oneline
8add7da Add initial content for module mod1.

$ cd ../mod2

mod2 ((7c2584f...))
$ git log --oneline
7c2584f update of info file
07f58e0 Add initial content for module mod2.
```

Specifically, notice that you don't have the latest commits, just the commits up to the time when the submodules were added in to the superproject you just cloned. This is a unique and important difference when working with submodules. Projects that contain submodules retain a *memory* of the commit that was active or used when the repository was added to the project as a submodule.

Also, if you look at what branch is active on the submodules, you can see that there isn't an active branch. The checked-out commit, which was current when the submodule was added, is the currently active *detached HEAD*. This is not as bad as it sounds. It simply means that rather than pointing to a specific branch reference, Git is pointing to a specific revision.

```
$ cd mod1; git branch
* (HEAD detached at 8add7da)
  master

$ cd ../mod2; git branch
* (HEAD detached at 7c2584f)
  master
```

Your prompt for mod2 may look something like this:

```
 <local path to mod>/mod2 ((7c2584f...))
```

This is an important point about submodules: they are tied initially to the same commit that was chosen when they were added to a container project. However, the repository for each of the submodules is still a separate Git repository that can have updates beyond when it was added as a submodule.

Because you know that updates have been made to the Git projects that compose the submodules you're using, this leads to the question of how you update your submodules to get the latest content. And, once the submodules are updated to a new commit, you have the added question of how you update your container project to ensure it records which commits its submodules now point to. There's also the question of how you can easily perform these kinds of operations across multiple submodules if you have more than one.

Let's look at an answer to that last question first.

Processing Multiple Submodules

As you've already seen, working with submodules is non-trivial. Furthermore, the level of complication can scale up when you are trying to manage multiple submodules. Luckily, Git includes a subcommand called *foreach* as part of the submodule command that simplifies doing the same operation across multiple submodules. The syntax for using this command is pretty straightforward.

```
git submodule [--quiet] foreach [--recursive] <command>
```

In this case, <command> can be whatever command you would like to run against the submodules, and it can be followed by additional arguments or options that are specific to that command. Using a git command as an example, you could use this functionality to see the logs of each submodule.

```
$ git submodule foreach git log --oneline
Entering 'mod1'
8add7da Add initial content for module mod1.
Entering 'mod2'
7c2584f update of info file
07f58e0 Add initial content for module mod2.
```

If you add the --quiet option, then the lines that say "Entering '<mod *name*>' are omitted from the output. The --recursive option is only needed if you have nested submodules—that is, submodules under your submodules.

Git also provides several variables populated with information that you can use when constructing commands. Those variables are as follows:

- $name—the name of the submodule
- $path—the path of the submodule relative to the superproject
- $sha1—the current SHA1 value of the submodule as recorded in the superproject
- $toplevel—the absolute path to the superproject

As an example of using these variables, you could construct a simple command to show the name of the module and the current SHA1 value that the superproject knows about.

```
$ git submodule --quiet foreach 'echo $path $sha1'
mod1 8add7dab652c856b65770bca867db2bbb39c0d00
mod2 7c2584f768973e61e8a725877dc317f7d2f74f37
```

Now, equipped with this option to process multiple submodules, let's return to how you can handle updates with submodules.

Updating Submodules from Their Remotes

If the remote repository that a submodule is based on has been updated, there are multiple approaches you can take to updating your submodules. (Note that I'm talking about the original project that the submodule was cloned from, not the superproject. This is the remote that shows up when you change into the submodule's directory and run git remote -v.) The approaches you can take are as follows:

- You can switch into each submodule, check out a branch (if needed), and do a pull or a fetch and merge.

  ```
  $ cd mod1
  $ git checkout <branch>
  $ git pull
  Updating 8add7da..a76a3fd
  Fast-forward
  ```

```
 mod1.info | 1 +
 1 file changed, 1 insertion(+)
```

- You can use the recurse-submodules option of git pull to update the contents of the submodules. This updates the default remote tracking branch in the submodule (usually origin/master). Then, you can go into each submodule and do a merge of the remote tracking branch into the local branch. (Again, this assumes that you've checked out a branch.)

  In the superproject, start by running the pull command with the recurse-submodules option:

  ```
  $ git pull --recurse-submodules
  Fetching submodule mod1
  From <remote path>/mod1
     8add7da..a76a3fd  master     -> origin/master
  Fetching submodule mod2
  From <remote path>/mod2
     7c2584f..cfa214d  master     -> origin/master
  Already up-to-date.
  ```

  Then, in the submodule, execute the merge:

  ```
  $ git merge origin/master
  Updating 8add7da..d05eb00
  Fast-forward
   mod1.info | 2 ++
   1 file changed, 2 insertions(+)
  ```

- You can use the update subcommand of the submodule command with the --remote option. In the superproject, run the following command:

  ```
  $ git submodule update --remote
  Submodule path 'mod1': checked out 'a76a3fd2470d21dcdca8a9671f39be383aae1ea1'
  Submodule path 'mod2': checked out 'cfa214db650ef5bcc7287323943d98b46d0a5354'
  ```

  If you only want to update a particular submodule, just add the submodule name to the end of the command.

  ```
  $ git submodule update --remote mod1
  ```

- You can iterate over each submodule using the foreach subcommand with operations to update the submodule. In the superproject, run the following command:

  ```
  $ git submodule foreach git pull origin master
  Entering 'mod1'
  From <remote path>/mod1
   * branch            master     -> FETCH_HEAD
  Already up-to-date.
  Entering 'mod2'
  From <remote path>/mod2
   * branch            master     -> FETCH_HEAD
  Updating 7c2584f..e9b2d79
  Fast-forward
   mod2.info | 2 ++
   1 file changed, 2 insertions(+)
  ```

NOTE *As I noted earlier in this chapter, information about submodules is stored in the .gitmodules file in the superproject.*

```
$ cat .gitmodules
[submodule "mod1"]
        path = mod1
        url = <remote path for mod1>
[submodule "mod2"]
        path = mod2
        url = <remote path for mod2>
```

The default branch for a submodule is assumed to be master. If you need to change that, you can do it through a simple git config command, such as the following:

```
$ git config -f .gitmodules submodule.mod2.branch testbranch
```

Here, the -f option is simply pointing to a different file—the .gitmodules file—and setting the value for the key triple submodule->mod2->branch. Afterward, your .gitmodules file looks like this:

```
$ cat .gitmodules
[submodule "mod1"]
        path = mod1
        url = C:/Users/bcl/submod/remote/mod1.git
[submodule "mod2"]
        path = mod2
        url = C:/Users/bcl/submod/remote/mod2.git
        branch = testbranch
```

Viewing Submodule Differences

Once you have updated your submodules to the latest pushed content, you will have differences between what's in your submodules and what your superproject has been referencing for the submodules. You can see these differences easily with the submodule status command.

```
$ git submodule status
+d05eb000ecb6cc1f00bc1b45d3e1cb6fb48e108d mod1 (heads/master)
+e9b2d790cf97ee43dc745d9996e07426e5570242 mod2 (heads/master)
```

Recall that the plus sign (+) on the front means that "the submodule's current version that's checked out is different from the SHA1 value in the containing repository."

You can also see this kind of difference by diffing.

```
$ git diff
diff --git a/mod1 b/mod1
index 8add7da..d05eb00 160000
--- a/mod1
+++ b/mod1
@@ -1 +1 @@
```

```
-Subproject commit 8add7dab652c856b65770bca867db2bbb39c0d00
+Subproject commit d05eb000ecb6cc1f00bc1b45d3e1cb6fb48e108d
diff --git a/mod2 b/mod2
index 7c2584f..e9b2d79 160000
--- a/mod2
+++ b/mod2
@@ -1 +1 @@
-Subproject commit 7c2584f768973e61e8a725877dc317f7d2f74f37
+Subproject commit e9b2d790cf97ee43dc745d9996e07426e5570242
```

There is a *submodule* option that you can add to the diff in these cases to make the output look more legible.

```
$ git diff --submodule
Submodule mod1 8add7da..d05eb00:
  > third update
  > update info file
Submodule mod2 7c2584f..e9b2d79:
  > update 3 to info file
  > update 2 to info file
```

Superproject versus Submodules

You've now updated your submodules to the latest content from their respective remote repositories. However, you haven't updated the original references (SHA1 values of the submodules) that were recorded in the superproject when you originally added the submodules. This can be a problem.

If you look at a status right now in the superproject, you'll see that Git knows that things have been updated in the submodules.

```
$ git status
On branch master
Your branch is up-to-date with 'origin/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:   mod1 (new commits)
        modified:   mod2 (new commits)

Submodules changed but not updated:

* mod1 8add7da...d05eb00 (2):
  > third update
  > update info file

* mod2 7c2584f...e9b2d79 (2):
  > update 3 to info file
  > update 2 to info file

no changes added to commit (use "git add" and/or "git commit -a")
```

Notice that in this case, Git treats the submodule directories like changed files. Also, on the last line of output, you can see that Git expects you to stage and commit the updates to the submodule information if you want it to point to new SHA1 values (new commits) for the submodules.

This is key to updating information for submodules. In the superproject, you have to stage and commit the information that Git is tracking about the submodules. Otherwise, bad things can happen because these are out of sync.

Let's look at an example. Because you haven't yet committed the changes relating to the submodules in the superproject, the superproject still thinks the submodules should be pointing to their old locations. Using a technique with foreach that you saw earlier, you can easily see this, as shown in the following example.

```
$ git submodule foreach 'echo $name $sha1'
Entering 'mod1'
mod1 8add7dab652c856b65770bca867db2bbb39c0d00
Entering 'mod2'
mod2 7c2584f768973e61e8a725877dc317f7d2f74f37
```

If you go into your submodules, you can see they've been updated and you can see those SHA1 values are references to past points in the histories.

```
$ cd mod1

$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean

$ git log --oneline
d05eb00 third update
a76a3fd update info file
8add7da Add initial content for module mod1.

$ cd ..
$ cd mod2

$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean

$ git log --oneline
e9b2d79 update 3 to info file
cfa214d update 2 to info file
7c2584f update of info file
07f58e0 Add initial content for module mod2.
```

Superproject reference is here

Superproject reference is here

Given that the superproject has old references, if you run the submodule update command (without the --remote option), this tells Git to update the submodules to the references (SHA1 values) that are current in the superproject. This operation is commonly used when trying to bring submodules up to date with a superproject.

```
$ git submodule update
Submodule path 'mod1': checked out '8add7dab652c856b65770bca867db2bbb39c0d00'
Submodule path 'mod2': checked out '7c2584f768973e61e8a725877dc317f7d2f74f37'
```

After this, you can see that you have back-leveled each submodule! This is probably not what you intended.

```
$ cd mod1

$ git status
HEAD detached at 8add7da
nothing to commit, working directory clean

$ git log --oneline
8add7da Add initial content for module mod1.

$ cd ..

$ cd mod2
$ git status
HEAD detached at 7c2584f
nothing to commit, working directory clean

$ git log --oneline
7c2584f update of info file
07f58e0 Add initial content for module mod2.
```

The Problem with Submodules

The previous example illustrates a fundamental issue and source of problems with using submodules: trying to keep the submodule references in the superproject in sync with the submodules, and vice versa.

Notice that if you now do a submodule status check, it indicates that everything is in sync (no plus signs (+) on the front). Everything is, but you've just back-leveled your submodules.

```
$ git submodule status
 8add7dab652c856b65770bca867db2bbb39c0d00 mod1 (8add7da)
 7c2584f768973e61e8a725877dc317f7d2f74f37 mod2 (7c2584f)
```

As another example, if these references are out of sync and that inconsistency is pushed to the remote for the superproject, then other users who pull that version of the superproject can end up back-leveling their submodules, even if they've updated their superproject before.

Updating the Submodule References

So, what do you have to do to keep the submodule references in sync with the submodules? Working from the superproject, let's go back to where you have the latest updates in the submodules.

```
$ git submodule update --remote
Submodule path 'mod1': checked out 'd05eb000ecb6cc1f00bc1b45d3e1cb6fb48e108d'
Submodule path 'mod2': checked out 'e9b2d790cf97ee43dc745d9996e07426e5570242'
```

The submodule status tells you that you have newer content checked out in the submodules versus the references the superproject knows about—again.

```
$ git submodule status
+d05eb000ecb6cc1f00bc1b45d3e1cb6fb48e108d mod1 (heads/master)
+e9b2d790cf97ee43dc745d9996e07426e5570242 mod2 (heads/master)
```

If you run a status command (the short version this time), you can see Git telling you that it knows the two submodules have been modified (just like a changed file).

```
$ git status -s
 M mod1
 M mod2
```

Now, you can simply do an add and a commit to update your superproject with the latest references for the submodules.

```
$ git 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:   mod1
        modified:   mod2

Submodule changes to be committed:

* mod1 8add7da...d05eb00 (2):
  > third update
  > update info file

* mod2 7c2584f...e9b2d79 (2):
  > update 3 to info file
  > update 2 to info file
```

You can then commit the updates into the superproject's repository.

```
$ git commit -m "update submodules to latest content"
[master 7e4e525] update submodules to latest content
 2 files changed, 2 insertions(+), 2 deletions(-)
```

Afterward, the submodule status shows you that the superproject and the submodules are in sync.

```
$ git submodule status
 d05eb000ecb6cc1f00bc1b45d3e1cb6fb48e108d mod1 (heads/master)
 e9b2d790cf97ee43dc745d9996e07426e5570242 mod2 (heads/master)
```

And, likewise, if you run the update command, there is no updating for Git to do.

```
$ git submodule update
```

Of course, the final step is to push the changes to the superproject over to the remote side. Otherwise, other users will not get those changes and you risk back-leveling again the next time a pull operation is done.

```
$ git push
Counting objects: 2, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (2/2), 338 bytes | 0 bytes/s, done.
```

```
Total 2 (delta 0), reused 0 (delta 0)
To <remote path>/main.git
   2745a27..7e4e525  master -> master
```

Updating Submodules When the Superproject Is Updated

What if you are using submodules and someone else updates the superproject, including updated submodule content? The solution is fairly simple thanks to an option that you saw earlier for pull: recurse-submodules. You can use the same operation again to get the updates into your local environment.

```
$ git pull --recurse-submodules
remote: Counting objects: 6, done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 6 (delta 2), reused 0 (delta 0)
Unpacking objects: 100% (6/6), done.
From C:/Users/bcl/submod/remote/main
   2745a27..5d1e722  master     -> origin/master
Fetching submodule mod1
From C:/Users/bcl/submod/remote/mod1
   a76a3fd..7e72f3c  master     -> origin/master
Fetching submodule mod2
From C:/Users/bcl/submod/remote/mod2
   cfa214d..e9b2d79  master     -> origin/master
Updating 2745a27..5d1e722
Fast-forward
 mod1 | 2 +-
 mod2 | 2 +-
 2 files changed, 2 insertions(+), 2 deletions(-)
```

However, this operation does not check out the updated references in your submodules. Your submodules are still registering the previous commits as current. You can see this when you run the status and log commands against them.

```
$ git submodule status
+a76a3fd2470d21dcdca8a9671f39be383aae1ea1 mod1 (heads/master)
+cfa214db650ef5bcc7287323943d98b46d0a5354 mod2 (heads/master)

$ cd mod1; git log --oneline
a76a3fd update info file
8add7da Add initial content for module mod1.

$ cd ../mod2; git log --oneline
cfa214d update 2 to info file
7c2584f update of info file
07f58e0 Add initial content for module mod2.
```

To get the latest commits registered and finish the update, you can just run the submodule update command to check out the updated references from the submodules.

```
$ cd ..; git submodule update
Submodule path 'mod1': checked out '7e72f3c96b19d7b6db38538e91d673e8249d418e'
Submodule path 'mod2': checked out 'e9b2d790cf97ee43dc745d9996e07426e5570242'
```

Afterward, your status and logs are consistent with the latest updates.

```
$ git submodule status
 7e72f3c96b19d7b6db38538e91d673e8249d418e mod1 (remotes/origin/HEAD)
 e9b2d790cf97ee43dc745d9996e07426e5570242 mod2 (remotes/origin/HEAD)

$ git log --oneline mod1
5d1e722 update 5 to mod1
482cf2f added new change in mod1
7e4e525 update submodules to latest content
2745a27 Add submodules mod1 and mod2

$ git log --oneline mod2
7e4e525 update submodules to latest content
2745a27 Add submodules mod1 and mod2

$ cd mod1; git log --oneline
7e72f3c update 5
2d25d0a another update
d05eb00 third update
a76a3fd update info file
8add7da Add initial content for module mod1.

$ cd ../mod2; git log --oneline
e9b2d79 update 3 to info file
cfa214d update 2 to info file
7c2584f update of info file
07f58e0 Add initial content for module mod2.
```

Of course, you can also pull (or fetch and merge) the code separately in each submodule and the superproject.

Pushing Changes from Submodules

Just as with any other aspect of using submodules, when changes that are to be pushed are made in the submodules themselves, there has to be coordination with the submodules and the superproject. When changes are pushed in a submodule, the references in the superproject also need to be pushed, and vice versa. Otherwise, you can get into those out-of-sync states again where your superproject thinks your current commit in the submodule should be in one place and the submodule thinks it should be in another. And, as I have already alluded to, if this out-of-sync condition is pushed into the remote, when other users clone or pull the superproject, they end up with the same out-of-sync condition and may not even realize it at first. Or worse, their local Git environment may be back-leveled.

Luckily, Git includes an option for push that can do some checking to enforce that everything is in sync: *recurse-submodules*. The recurse-submodules option takes two arguments, check and on-demand, that can be useful to you in this case.

The *check* argument tells the push command to verify that, in each submodule where code has been committed, the commit has also been pushed to at least one remote associated with the submodule. If not, it aborts the push and exits with a non-zero return code.

Here's what that might look like. Suppose you make an update in the submodule *mod1* and commit it (but don't push it):

```
$ cd mod1
$ echo "update 5" >> mod1.info

$ git commit -am "update 5"
[master 7e72f3c] update 5
 1 file changed, 1 insertion(+)
```

Going back to the superproject, you see the expected status that mod1 has changed.

```
$ cd ..

$ git status
On branch master
Your branch is up-to-date with 'origin/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:   mod1 (new commits)

Submodules changed but not updated:

* mod1 2d25d0a...7e72f3c (1):
  > update 5

no changes added to commit (use "git add" and/or "git commit -a")
```

You can now commit the change to the superproject's submodule information.

```
$ git commit -am "update 5 to mod1"
[master 5d1e722] update 5 to mod1
 1 file changed, 1 insertion(+), 1 deletion(-)
```

If we then try to push it and tell Git to check if all updates in the submodules have been pushed, Git catches that our change hasn't been pushed in the submodule and aborts the push.

```
$ git push --recurse-submodules=check
The following submodule paths contain changes that can
not be found on any remote:
  mod1
Please try
        git push --recurse-submodules=on-demand
or cd to the path and use
        git push
```

```
to push them to a remote.
fatal: Aborting.
fatal: The remote end hung up unexpectedly
```

The *on-demand* argument tells the push command to try pushing any commits that need to be pushed for the submodules at that point. If Git isn't successful in pushing something in a submodule, it aborts the push and exits with a non-zero return code.

Keeping with the previous example, if you change the *check* option to the *on-demand* option, Git tries to push the un-pushed change in the submodule for you (which it can do in this case).

```
$ git push --recurse-submodules=on-demand
Pushing submodule 'mod1'
Counting objects: 3, done.
Writing objects: 100% (3/3), 272 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To C:/Users/bcl/submod/remote/mod1.git
   2d25d0a..7e72f3c  master -> master
Counting objects: 2, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (2/2), 247 bytes | 0 bytes/s, done.
Total 2 (delta 1), reused 0 (delta 0)
To C:/Users/bcl/submod/remote/main.git
   482cf2f..5d1e722  master -> master
```

Submodules and Merging

By now, you should understand that dealing with submodules is all about keeping the submodules in sync with the submodule reference in the superproject. You also need to keep this overall model in mind if you run into a merge conflict when updating something in a submodule. In that case, you want to use the usual processes (as I discuss in Chapter 9) to resolve the merge conflicts, but then make sure to update the submodule reference in the superproject so it now points to the SHA1 value of the commit that contains the fixed merge.

Essentially, you can map out the process of dealing with a merge commit in a submodule as follows:

1. Change into the submodule and resolve the merge in the most appropriate way.
2. Change back to the superproject.
3. Verify that the expected values of the submodule updates match the superproject's references.
4. Stage (add) the updated submodule reference.
5. Commit to finish the merge.

Unregistering a Submodule

Finally, what happens if you want to *unregister* a submodule? To do this, you can use the *deinit* subcommand to the submodule command. When you use deinit, it removes the reference to the submodule from the superproject and removes the working tree from the subdirectory. If the working tree has modifications, you need to use the *--force* option to force the accompanying removal.

TIP *Here is a summary of the basic rules for dealing with submodules and superprojects:*

If you update something in a submodule, follow these steps:

1. *In the submodule directory, commit and push it out to the submodule's remote.*
2. *Go back to the superproject. The superproject should show that that particular submodule area has changed—almost like a file in the repository with the submodule name.*
3. *Stage and commit that changed area (submodule name) in the superproject to ensure that the superproject points to the updated commit in the submodule.*
4. *Push out that change in the superproject to the superproject's remote. This ensures that anyone cloning or pulling the superproject gets a version that points to the latest updates in the submodules.*

If you pull an update of the superproject, follow these steps:

1. *Ensure that you have also pulled the latest versions of the submodules (using the recurse-submodules option or foreach subcommand, or by pulling each area).*
2. *In the superproject, run the submodule update to check out the commit in the submodule that corresponds to the submodule references in the superproject.*

The need to manually update submodules and superproject references to submodules to always keep them in sync—and avoid back-leveling—presents a significant challenge when using submodules. Another kind of functionality is available in Git that provides a similar working model without the worry of trying to keep things synchronized: subtrees. You'll look at subtrees in the next section.

SUBTREES

The subtree functionality in Git provides another way to incorporate subprojects into your main project. In this case, each subproject is incorporated into a subdirectory.

With submodules, you maintained links from the superproject to the submodules. With subtrees, there are no special links or module files that have to be synchronized. Instead, the projects are just copied into subdirectories. They travel with the superproject.

As a development analogy, using a submodule is like linking to a particular version of a library that your project is dependent on. Using a subtree is like taking a copy of that library's source code and adding or including it in your project's directory tree. The advantage here is that users do not have to worry about keeping reference information like gitmodules files in sync. The disadvantage is that

you have additional size and scope tacked on to your superproject and you are no longer using a truly separate project—you're maintaining a private copy.

The syntax of the subtree command looks like this:

```
git subtree add   -P <prefix> <commit>
git subtree add   -P <prefix> <repository> <ref>
git subtree pull  -P <prefix> <repository> <ref>
git subtree push  -P <prefix> <repository> <ref>
git subtree merge -P <prefix> <commit>
git subtree split -P <prefix> [OPTIONS] [<commit>]
```

Note that this is another Git command that has multiple subcommands. Also note that each subcommand takes a <prefix> argument. You can think of the prefix argument as specifying the name or path of the relative subdirectory where the project exists as a subtree.

Figure 14-3 shows a way to think about the subtree setup. Note, however, that when you add a subproject, you are typically adding a particular branch.

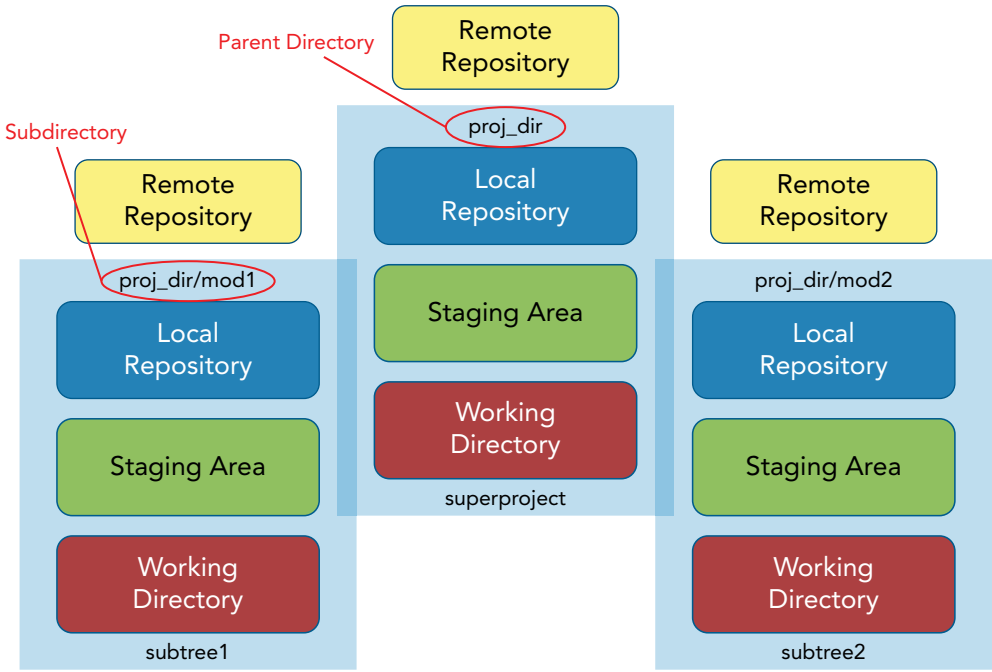


FIGURE 14-3: Illustration of a subtree layout

As an example of using the subtree command, let's look at how to add a project as a subtree.

Adding a Project as a Subtree

In its most basic form, adding a subproject as a subtree simply requires specifying a prefix, the remote path to the repository, and, optionally, a branch. Suppose you have cloned the remote project *myproj* down from a remote on your system.

```
$ git clone ../remotes/myproj.git myproject
Cloning into 'myproject'...
done.
```

This project contains three files.

```
$ cd myproject
$ ls
file1.txt  file2.txt  file3.txt
```

In the set of remotes that are available to you, you also have a project named *subproj*:

```
~/subtrees/remotes$ ls -la subproj.git
total 32
drwxr-xr-x  11 dev  staff   374B Aug  2 20:58 ./
drwxr-xr-x   4 dev  staff   136B Aug  2 20:59 ../
-rw-r--r--   1 dev  staff    23B Aug  2 20:58 HEAD
drwxr-xr-x   2 dev  staff    68B Aug  2 20:58 branches/
-rw-r--r--   1 dev  staff   164B Aug  2 20:58 config
-rw-r--r--   1 dev  staff    73B Aug  2 20:58 description
drwxr-xr-x  11 dev  staff   374B Aug  2 20:58 hooks/
drwxr-xr-x   3 dev  staff   102B Aug  2 20:58 info/
drwxr-xr-x   9 dev  staff   306B Aug  2 20:58 objects/
-rw-r--r--   1 dev  staff    98B Aug  2 20:58 packed-refs
drwxr-xr-x   4 dev  staff   136B Aug  2 20:58 refs/
```

Now, you add *subproj* with the master branch as a subtree to myproject.

```
~/subtrees/local$ cd myproject
~/subtrees/local/myproject$ git subtree add --prefix \
subproject ~/subtrees/remotes/subproj.git master
git fetch /Users/dev/subtrees/remotes/subproj.git master
warning: no common commits
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 5 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (5/5), done.
From /Users/dev/subtrees/remotes/subproj
 * branch            master     -> FETCH_HEAD
Added dir 'subproject'
```

If you look at the directory tree now, you see your new subdirectory underneath with its files.

```
~/subtrees/local/myproject$ ls
file1.txt   file2.txt   file3.txt   subproject/
```

```
~/subtrees/local/myproject$ ls subproject
subfile1.txt  subfile2.txt
```

And, if you look at the log, you can see the new commit where this project was added as a subtree, along with your comprehensive history.

```
~/subtrees/local/myproject$ git log --oneline
7d4f436 Add 'subproject/' from commit '906b5234f366bb2a419953a1edfb590aadc32263'
906b523 Add subfile2
5f7a7db Add subfile1
fada8bb Add file3
ef21780 Add file2
73e59ba Add file1
```

This is pretty straightforward. Let's take a look at another option that you can use with the add subcommand. First, you reset back to the place where you only have myproject without any subprojects.

```
$ git reset --hard HEAD~1
HEAD is now at fada8bb Add file3
~/subtrees/local/myproject$ git log --oneline
fada8bb Add file3
ef21780 Add file2
73e59ba Add file1
~/subtrees/local/myproject$ ls
file1.txt  file2.txt  file3.txt
```

Now to simplify things, we'll add a new remote reference to use in our commands:

```
~/subtrees/local/myproject$ git remote add sub_origin  ~/subtrees/remotes/subproj.git
```

When you add a subtree, by default, all of the project's history is also added in the subdirectory. To avoid adding all of the history, you can use a squash option. This squash option is similar to the squash option you used in the interactive rebasing functionality in Chapter 9. It compresses the history for the project that is being added into one commit.

Now, you add the subproject as a subtree again, this time using the squash option to compress the history.

```
~/subtrees/local/myproject$ git subtree add --prefix subproject --squash \
 sub_origin master

git fetch sub_origin master
From /Users/dev/subtrees/remotes/subproj
 * branch            master     -> FETCH_HEAD
 * [new branch]      master     -> sub_origin/master
Added dir 'subproject'
```

Looking at your files, you have the same structure as before.

```
~/subtrees/local/myproject$ ls
file1.txt   file2.txt   file3.txt   subproject/
```

However, notice that your history here has a record now that indicates the squashed history:

```
$ git log --oneline
6b109f0 Merge commit 'f7c3147d6df0609745228cc5083bb6c7d0b07d1a' as 'subproject'
f7c3147 Squashed 'subproject/' content from commit 906b523
fada8bb Add file3
ef21780 Add file2
73e59ba Add file1
~/subtrees/local/myproject$ git log -2
commit 6b109f0d5540642218d442297569b498f8e12396
Merge: fada8bb f7c3147
Author: Brent Laster <bl2@nclasters.org>
Date:   Tue Aug 2 21:15:06 2016 -0400

    Merge commit 'f7c3147d6df0609745228cc5083bb6c7d0b07d1a' as 'subproject'

commit f7c3147d6df0609745228cc5083bb6c7d0b07d1a
Author: Brent Laster <bl2@nclasters.org>
Date:   Tue Aug 2 21:15:06 2016 -0400

    Squashed 'subproject/' content from commit 906b523

    git-subtree-dir: subproject
    git-subtree-split: 906b5234f366bb2a419953a1edfb590aadc32263
```

Updating a Subtree

If you later need to pull some changes into your subtree, you can use a similar version of the subtree command with pull.

```
$ git subtree pull --prefix subproject sub_origin master --squash
```

This pulls down the latest content from the remote into the subtree areas and squashes the history again. You can omit the squash option to avoid compressing the history, but using this option will likely simplify things by not including all of the history.

There is also a *git subtree merge* command that you can use to merge commits up to a desired point into a subproject denoted by the --prefix argument. The git subtree merge command can be used to merge local changes to a subproject, while git subtree pull reaches out to the remote to get changes.

> **NOTE** *In Git, there is also a merge strategy named* subtree. *It's worth pointing out here that the git subtree command (actually a script that's been incorporated into Git) is not the same thing as the subtree merge strategy.*
>
> *Git chooses the best strategy depending on the situation (refer to Chapter 9). By default, it uses the recursive strategy to merge two branches, the octopus strategy to merge more than two branches, and so on. The subtree merge strategy is designed to be used when two trees are being merged and one is a subtree (subdirectory) of the other. In that case, the subtree merge strategy tries to shift the subtrees to be at the same level in order to merge similar structures. For more information, search for* subtree *in the help page for merge.*

Using the Subtree Split Functionality

The split subcommand for git subtree can be used to extract a subproject's content into a separate branch. It extracts the content and history related to <prefix> and puts the resulting content at the root of the new branch instead of in a subdirectory.

Let's look at an example. Suppose I have the following structure in my superproject:

```
~/subtrees/local/myproject$ls
file1.txt   file2.txt   file3.txt   subproject/
```

Now I want to extract out the content and history related to my *subproject* subdirectory in my subtree. I could use a command like the following:

```
~/subtrees/local/myproject$ git subtree split --prefix=subproject \
--branch=split_branch
Created branch 'split_branch'
906b5234f366bb2a419953a1edfb590aadc32263
```

As output, Git prints out the SHA1 value for the HEAD of the newly created tree, and so I have a reference to work with for that HEAD if needed. If you look into the new branch, you see only the set of content from the subproject that was split out (as opposed to content from the superproject).

```
~/subtrees/local/myproject$ git checkout split_branch
Switched to branch 'split_branch'
~/subtrees/local/myproject$ ls
subfile1.txt  subfile2.txt
~/subtrees/local/myproject$ git log --oneline
906b523 Add subfile2
5f7a7db Add subfile1
```

Creating a New Project from the Split Content

Given that you can split out content from a subtree, it follows that you may want to transfer that split content into another project. As it turns out, this is very simple with Git: you just create a new, empty project and then pull the branch contents over into it.

Here's an example based on my previous example. First, you create a new Git project.

```
~/subtrees/local/myproject$ cd ~/
~$ mkdir newproj
~$ cd newproj
~/newproj$ git init
Initialized empty Git repository in /Users/dev/newproj/.git/
```

Then, you can just pull the contents of the branch that you pulled out into this new repository.

```
~/newproj$ git pull ~/subtrees/local/myproject split_branch
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 5 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (5/5), done.
From /Users/dev/subtrees/local/myproject
 * branch            split_branch -> FETCH_HEAD
```

You can see that you have the same content in your new project that matches what you split out in the old repository.

```
~/newproj$ ls
subfile1.txt  subfile2.txt
~/newproj$ git log --oneline
906b523 Add subfile2
5f7a7db Add subfile1
```

Subtree Push

The subtree command also supports a push subcommand. This command does a split followed by an attempt to push the split content over to the remote. To illustrate, the following command splits out the subproject directory and then pushes it to the sub_origin remote reference and into a new branch named *new branch*:

```
~/subtrees/local/myproject$ git subtree push --prefix=subproject sub_origin new_
branch
git push using:  sub_origin new_branch
Total 0 (delta 0), reused 0 (delta 0)
To /Users/dev/subtrees/remotes/subproj.git
 * [new branch]      906b5234f366bb2a419953a1edfb590aadc32263 -> new_branch
~/subtrees/local/myproject$
```

SUMMARY

In this chapter, I've covered ways to work with multiple instances of working areas and repositories in your local environment. I also covered worktrees that allow you to work on multiple branches at the same time in different areas—all connected back to one local repository.

I discussed what submodules are—a type of linking to other projects from your original project (the superproject). I explained how the connection works and why submodule use is problematic. I then described another alternative for managing subprojects in subdirectories: subtrees.

Note that while I have discussed the options here for working with code in dependent repositories, a better approach in most situations would be to consume deliverables built by these other repositories as artifacts during the build or deployment process. You should limit your use of submodules and subtrees to when there are true source dependencies between repositories and you need that kind of close source connection. Of course, too many of these kinds of dependencies can also indicate a need to refactor code between repositories.

In the last chapter of this book, I'll look at how to extend the functionality of Git through its built-in mechanism for running programs before or after Git operations: Git hooks.

ABOUT CONNECTED LABS 10–12

There are three labs for this chapter to allow for focusing on each of the main topics: one for worktrees, one for submodules, and one for subtrees. A brief description of each lab follows.

About Connected Lab 10: Working with Worktrees

In this lab, you'll see how to work with the worktrees feature of Git. You'll get to create a worktree for a specific branch, see how to use it, and remove it.

About Connected Lab 11: Working with Submodules

This lab will give you some practice with submodules. You'll see how to add a repository as a submodule, make changes in it, and ensure that the containing project (the superproject) is updated.

About Connected Lab 12: Working with Subtrees

This lab demonstrates some of the basic operations in Git when working with subtrees. Like the submodule lab, you'll see how to add a repository as a subtree, and make changes in it.

As well you'll get a chance to split a subtree into a separate branch and then pull that content into a separate project.

Connected Lab 10

Working with Worktrees

In this lab, you'll get some experience with worktrees. For this and the subsequent labs, I have split the calc2 repository that you used in the last lab into three separate projects.

PREREQUISITES

This lab requires that you have Internet access and have completed Connected Lab 8: Setting up a GitHub Account and Cloning a Repository. You will be working in a new directory.

STEPS

1. For this lab, you need access to your GitHub account that you set up in Connected Lab 8. I have split up the calc2 project you used in Connected Lab 9 into three separate projects: *super_calc*, a version of the calc2 project with only the master and feature branches; *sub_ui*, a separate repository consisting of only the content of the ui branch split out from the calc2 project; and *sub_docs*, a separate repository consisting of only the content of the docs branch split out from the calc2 project. Log in to your GitHub account and fork the three projects from the following listed locations. (As a reminder, the fork button is in the upper-right corner of the pages.) This will prepare your area on GitHub for doing this lab, as well as Connected Labs 11 and 12.

   ```
   https://github.com/professional-git/super_calc.git
   https://github.com/professional-git/sub_ui.git
   https://github.com/professional-git/sub_docs.git
   ```

2. In a new directory, clone down the super_calc project that you forked in step 1, using the following command:

   ```
   $ git clone https://github.com/<your github userid>/super_calc.git
   ```

3. Now, change into the cloned directory—super_calc.

   ```
   $ cd super_calc
   ```

4. In this case, you want to work on both the master branch and the features branch at the same time. You can work on the master branch in this directory, but you need to create a separate working tree (worktree) for working on the features branch. You can do that with the worktree add command, passing the -b to create a new local branch off of the remote tracking branch.

```
$ git worktree add -b features ../super_calc_features origin/features
```

5. Change into the new subdirectory with the new worktree. Note that you are on the features branch. Edit the calc.html file and update the line in the file surrounded by <title> and </title>. The process is described below.

```
$ cd ../super_calc_features
```

Edit calc.html and change

```
<title>Calc</title>
```

to

```
<title> github_user_id's Calc</title>
```

substituting in your GitHub user ID for "github_user_id".

6. Save your changes and commit them back into the repository.

```
$ git commit -am "Updating title"
```

7. Switch over to your original worktree.

```
$ cd ../super_calc
```

8. Look at what branches you have there.

```
$ git branch
```

9. Note that you have the features branch you created for the other worktree. Do a log on that branch; you can see your new commit just as if you had done it in this worktree.

```
$ git log --oneline features
```

10. You no longer need your separate worktree. However, before you remove it, take a look at what worktrees are currently there.

```
$ git worktree list
```

11. You can now remove the worktree. First, remove the actual directory; then use the prune option to get rid of the worktree reference.

```
$ rm -rf ../super_calc_features
$ git worktree prune
```

Connected Lab 11

Working with Submodules

In this lab, you'll get some experience working with submodules.

PREREQUISITES

This lab requires that you have Internet access and have completed at least the first two steps in Connected Lab 10, where you forked the various split projects of the original calc2 project into your area in GitHub and cloned the super_calc project down to your local system.

STEPS

1. Start out in the super_calc directory for the super_calc repository that you cloned from your GitHub fork in Connected Lab 10. You're going to add another repository as a submodule to super_calc.
2. Add the sub_ui repository as a submodule to the super_calc project by running this command:

   ```
   $ git submodule add https://github.com/<your github userid>/sub_ui sub_ui
   ```

3. This adds sub_ui as a submodule to your super_calc repository. In the process, Git clones down the repository into the subdirectory and also creates and stages a .gitmodules file to map the connection with the super_calc project. Look at the directory listing to see the new subdirectory. Then look at the status to see the staged .gitmodules file. Finally, display the contents of the .gitmodules file to see what's in there. Run the following commands from the super_calc subdirectory:

   ```
   $ ls
   $ git status
   $ git show :.gitmodules
   ```

4. Now you need to commit and push the staged submodule mapping and data to your local and remote repositories. Run the following commands:

```
$ git commit -m "Add submodule sub_ui"
$ git push
```

5. Now you can clone a new copy of this project with the submodule. Change to a higher-level directory and clone a copy of the project down as super_calc2.

```
$ cd ..
$ git clone https://github.com/<your github userid>/super_calc super_calc2
```

6. Change into the super_calc2 directory and look at what's in the sub_ui subdirectory. Run the submodule status command to see what the status of the submodule is.

```
$ cd super_calc2
$ ls sub_ui
$ git submodule status
```

7. Notice the hyphen (-) in front of the SHA1 value. This indicates that the submodule has not been initialized yet relative to the super project. You could have done this at clone time using the --recursive option. However, because you didn't, you need to use the update --init subcommand for the submodule operation, as follows:

```
$ git submodule update --init
```

8. Git clones the sub_ui code into the submodule. Look at the sub_ui subdirectory to see the contents, and then run the submodule status command again.

```
$ ls sub_ui
$ git submodule status
```

This time, you see a space at the beginning (instead of the minus sign) to indicate the submodule has been initialized.

9. Now, you need to make a simple update to the code in the submodule. Change into the sub_ui subdirectory (cd sub_ui) and edit the calc.html file there as follows: change the line

```
<title>Advanced Calculator</title>
```

in the file to

```
<title> your_name_here Advanced Calculator</title>
```

substituting your actual name for "your_name_here". Save your changes.

10. Commit your changes into the submodule.

```
$ git commit -am "Update title"
```

11. Do a quick log command and note the SHA1 value associated with the commit you just made.

```
$ git log --oneline
```

12. Change back to the super_calc2 project (up one level) and run a submodule status command.

```
$ cd ..
$ git submodule status
```

13. Note that the submodule SHA1 reference now points to your latest commit in the submodule, but there is a plus sign (+) at the front of the reference. This indicates that there are changes in the submodule that have not yet been committed back into the super project. Run a git status command to get another view of what's changed for the super project.

```
$ git status
```

14. The status command gives you much more information about what's changed. It tells you that the sub_ui module has been changed and is not updated or staged for commit. To complete the update, you need to stage and commit the sub_ui data. You can then push it out to your GitHub remote repository. To complete the process, execute the commands below.

```
$ git commit -am "Update for submodule sub_ui"
$ git push
```

15. Now that you've updated the submodule and the supermodule, everything is in sync. Run a git status and a git submodule status command to verify this.

```
$ git status
$ git submodule status
```

Connected Lab 12

Working with Subtrees

In this lab, you'll get some experience working with subtrees.

PREREQUISITES

This lab requires that you have Internet access and have completed at least the first two steps in Connected Lab 10, where you forked the various split projects of the original calc2 project into your area in GitHub and cloned the super_calc project down to your local system.

STEPS

1. Start out in the super_calc directory for the super_calc repository that you cloned from your GitHub fork in Connected Lab 10. You're going to add another repository as a subtree to super_calc.

2. To add the repository, use the following command:

   ```
   $ git subtree add -P sub_docs --squash https://github.com/<your github user id>/sub_docs master
   ```

   Even though you don't have much history in this repository, you used the --squash command to compress it. Note that the -P stands for prefix, which is the name your subdirectory gets.

3. Look at the directory structure; note that the sub_docs subdirectory is there under your super_calc project. Also, if you do a git log, you can see where the subproject was added and the history squashed.

   ```
   $ ls sub_docs
   $ git log --oneline
   ```

   Note that there is only one set of history here because there is only one project effectively—even though we have added a repository as a subproject.

4. Now, you will see how to update a subproject that is included as a subtree when the remote repository is updated. First, clone the sub_docs project down into a different area.

```
$ cd ..
$ git clone https://github.com/<your github user id>/sub_docs sub_docs_temp
```

5. Change into the sub_docs_temp project, and create a simple file. Then stage it, commit it, and push it.

```
$ cd sub_docs_temp
$ echo "readme content" > readme.txt
$ git add .
$ git commit -m "Adding readme file"
$ git push
```

6. Go back to the super_calc project where you have sub_docs as a subtree.

```
$ cd ../super_calc
```

7. To simplify future updating of your subproject, add a remote reference for the subtree's remote repository.

```
$ git remote add sub_docs_remote https://github.com/<your github user id>/sub_
docs
```

8. You want to update your subtree project from the remote. To do this, you can use the following subtree pull command. Note that it's similar to your add command, but with a few differences:

   - You used the long version of the prefix option.
   - You are using the remote reference you set up in the previous step.
   - You don't have to use the squash option, but you add it as a good general practice.

```
$ git subtree pull --prefix sub_docs sub_docs_remote master --squash
```

   Because this creates a merge commit, you will get prompted to enter a commit message in an editor session. You can add your own message or just exit the editor.

9. After the command from step 8 completes, you can see the new README file that you created in your subproject sub_docs. If you look at the log, you can also see another record for the squash and merge_commit that occurred.

```
$ ls sub_docs
$ git log --oneline
```

10. Changes you make locally in the subproject can be promoted the same way using the subtree push command. Change into the subproject, make a simple change to your new README file, and then stage and commit it.

```
$ cd sub_docs
$ echo "update" >> readme.txt
$ git commit -am "update readme"
```

11. Change back to the directory of the super_project. Then use the subtree push command below to push back to the super project's remote repository.

```
$ cd ..
$ git subtree push --prefix sub_docs sub_docs_remote master
```

Note the similarity between the form of the subtree push command and the other subtree commands you've used.

12. The next few steps show you how to take a subproject, put it onto a different branch, and then bring that content into a separate repository. First, use the subtree split command to take the content from the sub_docs subproject and put it into a branch named *docs_branch* in the super_calc area.

```
$ git subtree split --prefix=sub_docs --branch=docs_branch
```

13. Look at the history for the new docs_branch. You can see all of the content that you have in the sub_docs project.

```
$ git log --oneline docs_branch
```

14. Create a new project into which you can transfer the docs_branch content.

```
$ cd ..
$ mkdir docs_proj
$ cd docs_proj
$ git init
```

15. Finally, use the git pull command in a slightly different context to pull over that content into the master branch of your new project.

```
$ git pull ../super_calc docs_branch:master
```

16. Do a git log of the master branch in the new project to see the copied content.

```
$ git log --oneline master
```

15

Extending Git Functionality with Git Hooks

WHAT'S IN THIS CHAPTER?

- Understanding Git hooks
- Installing hooks
- Preparing environments for hooks
- Learning details for each Git hook
- Exploring hooks written in bash, Groovy, Ruby, Perl, and Python

Now that I've covered the overall Git workflow and functionality in detail, let's look at one way you can extend and customize how Git works: hooks. *Hooks* are scripts or programs that run before or after (and in some cases during) a subset of operations in Git. There are hooks for local operations, such as commits or merges, and hooks for remote operations, such as when changes are pushed to the remote.

In some other source management systems, hooks may be called by other names, such as *triggers*. However, the concept is the same—a program or script runs when a certain event or operation happens. Here are some common uses for hooks:

- Sending e-mail or other notifications when a change is pushed to a repository
- Validating that certain conditions have been met before a commit
- Appending items to commit messages
- Checking the format or existence of certain elements in a commit message
- Updating content in the working directory after an operation
- Enforcing coding standards

INSTALLING HOOKS

By default, hooks exist in the hooks directory underneath your Git directory. Your Git directory is directly under your working directory unless you've overridden that location by setting a different value in the $GIT_DIR environment variable, or by passing a different value to --git-dir on the Git command line. As a result, the default setting means that the hooks reside in .git/hooks.

As of Git 2.9, you can set a different relative or absolute path for hooks using the core.hooksPath configuration value. This allows you to set a centralized path to find the hooks for multiple projects or even multiple users if everyone has access to the path. For the rest of this chapter, I will refer to the *hooks directory* to mean wherever you have set your hooks to be.

After cloning or using init to create a project, if you look in the .git/hooks directory, you see a list of files like the following:

```
myproject/.git/hooks$ ls
applypatch-msg.sample*     pre-push.sample*
commit-msg.sample*         pre-rebase.sample*
post-update.sample*        prepare-commit-msg.sample*
pre-applypatch.sample*     update.sample*
pre-commit.sample*
```

> **NOTE** *It is important to note that hooks are not cloned from a remote repository. This is by design because hooks that run on the remote (server) side may not be suitable for local use or may require special permissions that are not allowed on the user's system. Also, note that a pull or fetch does not update the hooks.*

Notice that all of these files end in .sample. The part of the filename before .sample is the actual hook name that Git expects. The .sample extension is there to allow the files to exist as sample hooks without actually being executed as hooks. In effect, the extension *disables* the hook. To enable any of these hooks, you need to remove the .sample extension. For example, to enable the commit-msg hook in this repository, you add whatever code or processing you want to use to the commit-msg.sample file and rename it as simply *commit-msg*. When Git verifies whether the commit-msg hook is enabled, it finds the file in the hooks directory with the expected name and executes the file. (Note that hook files also need to be executable, so it may be necessary to change the permissions for the file if it is not already executable.)

By default, the initial set of files in .git/hooks comes from the hooks directory of the template area where Git is installed (or where configuration points to the template). Running git init again on an existing directory picks up new content from the template area but doesn't overwrite changed content. This provides another option for creating new hooks and distributing them to existing repositories by re-running the init command. See the help page for git init for more information on how to specify the location of the template area to populate from.

UPDATING HOOKS

Because hooks are not updated as part of a clone, pull, or fetch, it can be challenging to get updated hooks into all repositories that need them. There are a few methods that you can use to do this:

- If everyone is using Git 2.9 or later, you can designate a commonly readable area for the hooks using the core.hooksPath configuration setting.
- You can store the actual hooks scripts as part of the project and then copy or symlink them to the .git/hooks directory. You can also create a script file and include it in the project to automate the copying or symlinking from the working directory to the hooks directory when the script is run.
- In a similar way, you can include a script with each project to copy updated hooks from a common area.
- If the hook is a new file, you can update it in the templates area, and then users can run the git init command again to pick up the new file.

COMMON HOOK ATTRIBUTES

Next, let's look at some of the attributes and behaviors that are shared across sets of hooks.

Hook Domain

Hooks can be targeted for execution on either the remote repository side (remote hooks) or the local repository side (local hooks). Local hooks are designed to run before or after local operations such as commit, checkout, rebase, and so on. Remote hooks are designed to run before or after operations on the remote side, such as push.

Here is the list of available local hooks:

- applypatch-msg
- pre-applypatch
- post-applypatch
- pre-commit
- prepare-commit-msg
- commit-msg
- post-commit
- pre-rebase
- post-checkout

- post-merge
- pre-push
- post-rewrite
- push-to-checkout

Here is the list of available remote hooks:

- pre-receive
- update
- post-receive
- post-update
- pre-auto-gc

Return Code to Control Workflow

In many cases, hooks are used not only to do additional processing, but also to serve as checks on whether or not operations should continue. In these cases, the hook checks some value or condition and exits with a return code to indicate whether or not the check passed or failed. A return code of zero indicates that all is good and the operation should continue. A non-zero return code indicates that something wasn't correct or that a check failed, and tells Git to abort the operation. Different non-zero values can be returned to indicate specific error states.

Working Directory Access

One of the important considerations when a hook runs is what directory it is running in. Git defaults to one of two places to run hooks, depending on whether the hook is executing in a bare repository or a non-bare repository.

> **NOTE** *As a reminder, a bare repository is one that does not have a corresponding staging area and working directory—it does not have a checked-out copy of a branch. Most commonly, you see it as a remote repository or a repository on the server side, where a checked-out working directory is not needed and doesn't make sense.*
>
> *Bare repositories only need the actual repository directory (the one that corresponds to .git in a local repository), so they are named as <name>.git, the directory that holds the actual repository.*
>
> *Non-bare repositories are the ones you are used to working with locally, where you have a checked-out branch in a working directory and a staging area. Those repositories are named with a directory (for the checked-out content) and then have the .git directory (or whatever is specified by the GIT_DIR setting or on the command line with the --git-dir option) as a subdirectory with the actual repository.*

When Git runs a hook, if it is executing in a non-bare repository, it first changes to the root directory of the working directory. For example, if the local environment is in /usr/home/myrepo and an operation there causes a hook to run from /usr/home/myrepo/.git/hooks, then Git uses /usr/home/myrepo as the directory to run the hook in.

If the hook is running in a bare repository, then Git uses the directory of the bare repository. For example, if the bare repository is in /usr/share/git/repos/myrepo.git and you are executing a hook in /usr/share/git/repos/myrepo.git/hooks, then Git uses /usr/share/git/repos/myrepo.git as the directory to run the hook in.

Environment Variables

Each hook has access to some environment variables that are set through Git. Nearly all hooks have access to GIT_DIR. For the reasons outlined in the previous section, this variable usually has a value of ".git" for local hooks and "." for remote hooks.

Hooks that work with the am or commit operations have values set for the date (GIT_AUTHOR_DATE), e-mail (GIT_AUTHOR_EMAIL), and username (GIT_AUTHOR_NAME), as set in the Git configuration. Some operations, such as those for rebasing or merging, also have a GIT_REFLOG_ACTION value corresponding to the operation being performed and whatever is written in the reflog.

HOOK DESCRIPTIONS

The rest of this chapter includes individual descriptions of the various hooks. The descriptions include what each hook does and what it is used for, information on bypassing (for those that can be bypassed), a description of the parameters the hook gets, and whether the hook can abort the operation.

For some of the hooks, I include example code that implements the basic hook functionality. These examples are written in several different languages, including shell, Ruby, Groovy, Python, and Perl, so you have a simple reference for each of them. The examples are contrived and not meant to include all functionality that a hook can provide or to be the best example of coding hooks. (These examples can also be downloaded from http://github.com/professional-git/hooks.)

Applypatch-msg

In Chapter 11, I talk about the git am command. The am command is designed to take a patch in an e-mail format and commit it into the local repository. The applypatch-msg hook is intended to operate on the proposed commit message for the apply part of the am operation.

Applypatch-msg takes a single parameter: the name of the temporary file that has the proposed commit message. A non-zero exit code from this hook aborts the apply part of the am operation. This hook can be used to verify the message file or even edit it in place.

Pre-applypatch

Pre-applypatch is another hook that works with the git am command. As I previously mentioned, the am command takes a patch, applies it, and commits the changes. The pre-applypatch hook is

called after the patch is applied, but before any changes are committed. In this way, it allows Git to verify that the results of applying the patch are as expected. For example, this could mean verifying that the patch did no harm, by doing some kind of testing or building.

This hook does not take any parameters. If the hook exits with a non-zero return code, then the updated code (code with the patch applied) is not committed—at least not as part of the am operation.

Post-applypatch

Post-applypatch is the last of the hooks intended for use with the am command. This hook is called after the patch is applied and committed. It is primarily used for sending notifications. However, it can also be used for launching some kind of processing to verify that what was committed was viable. However, it would probably make more sense to do that kind of check with a pre-apply hook, as I described in the previous section.

The post-applypatch hook does not take any parameters. Because it is run after the apply and commit are completed, it cannot influence the outcome of the operation.

Pre-commit

As its name implies, the pre-commit hook is executed before a commit. In fact, it is executed before Git even asks the user to enter a commit message or puts together a commit object.

Pre-commit can be bypassed with the --no-verify option to git commit. It takes no parameters. A non-zero exit code from this hook aborts the commit.

You can use the pre-commit hook to verify that what's about to be committed meets some condition or criteria and is okay to commit. An example script for this hook (written in Python) is included here. In this case, you don't want to allow any files to be committed that have a reference to a company's internal-use-only website (IUO_WEBSITE in the script). The script gets a list of staged files, and checks their contents for the disallowed string. If it finds the string, it prints an error message and exits with -1, thus failing the commit.

```
#!/usr/bin/env python

# Example pre-commit hook written in python
# (c) 2016 Brent Laster

import sys, os, subprocess

# define the string we want to check for
IUO_WEBSITE='http://internal.mycompany.com'
try:
             # Get the status from Git in the short form
             # Note: Could use -s instead of --porcelain
             #  but porcelain is guaranteed not to break
             #  backwards-compatibility between releases.
             status_output = subprocess.check_output(
                    ["git","status","-uno","--porcelain"],
```

```
                stderr=subprocess.STDOUT).decode("utf-8")

        # create a list of status lines
        status_list = status_output.splitlines()

        for status_line in status_list:
                # if the status line starts with A, then it is in the staging area
                if status_line.startswith('A'):
                        status, _, status_file = status_line.partition(" ")
                        staged_file = status_file.lstrip()
                        # git the relative path from Git in case the file
                        #  is in a subdirectory of the working directory
                        rpath = subprocess.check_output(
                              ["git","rev-parse","--show-prefix"],
                             stderr=subprocess.STDOUT).decode("ISO-8859-1")
                        relative_path = rpath.rstrip()

                        # construct the :<path> syntax needed for show
                        #  to dump the contents of the staged version
                        staged_fullpath = ":" + relative_path + staged_file

                        # Use the git show :<path> syntax to get the contents
                        #  of the actual staged version
                        staged_content = subprocess.check_output(
                              ["git","show",staged_fullpath],
                              stderr=subprocess.STDOUT).decode("ISO-8859-1")
                        # if we find the forbidden string, then abort the commit

                        if IUO_WEBSITE in staged_content:
                              tpath = relative_path + staged_file
                              print ("Error!",IUO_WEBSITE,"found in file",tpath)
                              print ("Commit disallowed due to error.")
                              sys.exit(-1)
          sys.exit(0)

except subprocess.CalledProcessError:
       print ("error!")
       sys.exit(-2)
```

Prepare-commit-msg

After the pre-commit hook checks and validates the content to be committed, the prepare-commit-msg hook is called. Its purpose is to do any additional editing or preparation of the commit message before it is brought up in an editor. (Note, however, that this hook is also called if you pass in a commit message with the -m option.)

The prepare-commit-msg hook takes a minimum of one and a maximum of three parameters, defined as follows:

- Parameter 1—The name of the file that contains the proposed commit message.
- Parameter 2—The type of operation that precipitates the message. The value can be one of the following:

    - message—If you passed a message to the commit using the -m option or the -F option, from a file
    - template—If you used the -t option or the commit.template configuration value was set
    - merge—If the commit is the result of a merge
    - squash—If the commit is the result of a squash
    - commit—If this is just a regular commit
- Parameter 3—A SHA1 value if you used the --amend, -c, or -C option. (-C allows you to reuse a commit message from the specified SHA1 value; -c does the same, but also invokes the editor.)

The prepare-commit-msg hook is not suppressed by the --no-verify option. If this hook returns a non-zero return code, then the commit operation aborts.

An example script for this hook (written in Groovy) is included here. In this example, there is user-defined text that must be included in every commit message. The user-defined text is defined by configuring the new setting *user.msg* in Git. If that configuration setting is not found, the hook exits and aborts the commit. If the message already contains the text, the hook simply proceeds. Otherwise, if we are in the master branch, or a branch that starts with "prod" or "ship", it appends the text to the commit message.

```
#!/usr/bin/env groovy

// Example prepare-commit-msg hook written in groovy
// (c) 2016 Brent Laster

import static java.lang.System.*

def argc = this.args.size()
def commitmsg_file_path, commit_type, commit_sha1

// Get explicit commit type if passed
commitmsg_file_path = this.args[0]
if (argc > 1)
       commit_type = this.args[1]
else
    commit_type = ''

if (argc > 2)
    commit_sha1 = this.args[2]
else
    commit_sha1 = ''

// See if we have user-defined message set
def iuo_msg = ["git", "config", "user.msg"].execute()
iuo_msg.waitFor()
def config_rc = iuo_msg.exitValue()
// If we don't have the configuration value set, then abort
if (config_rc!=0)
{
       println "Configuration setting not found for user.msg."
```

```
        println "Aborting commit!"
        exit 1
}
def msg = iuo_msg.text

// Read in an existing commit message
def File commit_file = new File(commitmsg_file_path)

// If the commit message already contains the value, then just continue
if (commit_file.text.find(msg))
{
println("Commit message already contains $msg - proceeding...")
exit 0
}

// Determine the branch
def output_branch = ["git", "symbolic-ref", "--short", "HEAD"].execute()
output_branch.waitFor()
def current_branch = output_branch.text.trim()

// If it matches the desired branch names, then append the custom message
if  (current_branch.matches(/^master$|^(prod|ship).*$/))
{
        println "Current branch $current_branch is a production branch."
        commit_file.append(msg)
}
exit 0
```

Commit-message

The commit-msg hook is called after the user enters a commit message. Its main purpose is to validate the commit message that the user has entered. For example, it can check the message for certain required information or to make sure the contents fit an expected corporate format. It can also be used to edit the commit message file in place—for example, to append unique information to each commit message before the commit is processed. This append functionality is used in the Gerrit Code Review tool, where the commit-msg hook plays a key role in appending a unique Change-Id to each commit message. Note that, depending on need and timing, the functionality I discuss in the "Prepare-commit-message" section can also apply here, after the message is entered.

This hook can be bypassed with the --no-verify option. The hook takes one argument: the name of the temporary file with the commit message in it. If the hook returns a non-zero return code, then the commit operation aborts.

The example that I include for the commit-msg hook is written as a bash shell script. This hook implements several checks around the commit message. First, it checks to see if the commit message is a certain minimum length. If the message is too short, the hook aborts the commit. Second, it checks to see if the message is the same as the previous commit. If it is, the hook also aborts the commit. Finally, it checks to see if a user-defined message is included in the commit message. Like the prepare-commit-msg hook, the message is expected to be defined in the Git configuration value, user.msg.

```
#!/usr/bin/env bash

# Example commit-msg hook written in bash
# (c) 2016 Brent Laster
```

```
# define our error messages
error_msg1="Error: Commit message is too small."
error_msg2="Error: Commit message cannot be the same as the previous commit message."
error_msg3="Error: Expected text not in commit message:"
abort_msg="Commit will not be allowed due to error."
minimum_length=25

# just exit if we don't have any arguments
[ "$1" ] || exit 1

# read the contents of the temp commit message file and get the length of it
contents=$(<"$1")
size=${#contents}

# if the message is too short, error out
if [ $size -le $minimum_length ]; then
       echo "$error_msg1" >&2
       echo "$abort_msg" >&2
       exit 1
fi

# if we have a previous revision, make sure we're not using the same message as the
last commit
if git rev-parse --verify HEAD >/dev/null 2>&1
then
    previous_log_msg=$(git show -s --format=%s)
       if [ "$previous_log_msg" == "$contents" ]; then
             echo "$error_msg2" >&2
             echo "$abort_msg" >&2
             exit 1
       fi
fi

# if our branch is master or starts with "prod" or "ship"
#  check to make sure we have the value defined in user.msg in the commit message
branch=$(git symbolic-ref --short HEAD)
if [[ $branch =~ ^master$|^(prod|ship).*$ ]]; then
       iuo_msg=$(git config user.msg)
       if ! grep -iqE "$iuo_msg" "$1"; then
             echo "$error_msg3" "$iuo_msg" >&2
             echo "$abort_msg" >&2
             exit 1
       fi
fi

# Redirect output to stderr.
exec 1>&2

exit 0
```

Post-commit

The post-commit hook is invoked after the commit-msg hook. It is primarily used for notification services, although you can also use it to launch a post-commit operation. For example, it could

launch some kind of continuous integration builds or testing at the level of the local repository to ensure the code is good before you push it over to the remote repository.

Post-commit doesn't take any parameters. Thus, you need to use some sort of git call in the hook to determine the latest SHA1 value to work against. There are a couple of calls you can use to do this, such as git log -1 HEAD or git rev-parse HEAD.

The post-commit hook's exit code doesn't affect the operation because the commit has already been done at this point. This hook, if it is active, can't be bypassed by the --no-verify option.

Note that a corresponding hook is available on the remote side: post-receive. This is a better choice for launching continuous integration processes or testing if you want to run them against changes committed and then pushed from all users.

The following example is written in perl. It verifies whether the checkout is for a branch that starts with *web*. If so, it checks out a copy of the files into a separate directory. The user is expected to set the value for the desired directory using a Git configuration setting, hooks.webdir.

```
#!/usr/bin/env perl

# Example post-commit hook written in perl
# (c) 2016 Brent Laster

use strict;
use warnings;

use autodie;
use File::Temp qw( tempfile );
use IPC::Cmd qw( run );


# if we are doing a commit from a branch named web* then
# point git to the website worktree and do a checkout -f to mirror files out
my $web_dir = 'git config hooks.webdir';
chomp($web_dir);
my $new_head_ref = 'git rev-parse --abbrev-ref HEAD';

# remove since git index doesn't exist here
delete $ENV{'GIT_INDEX_FILE'};
if (defined($web_dir) && ($new_head_ref =~ /^web.*$/)) {
       my $results = 'git --work-tree="$web_dir" --git-dir="$ENV{'GIT_DIR'}"
checkout -f';
}
```

Pre-rebase

As its name implies, the pre-rebase hook is called prior to a git rebase and before Git actually does any operations related to the rebasing. As such, this hook provides an opportunity to validate that the rebase should go through, issue a warning message, and so on.

The sample hook that comes with Git for pre-rebase has an extensive example of how the hook can be used. It prevents topic branches that have already been merged from being rebased (and thus merged again).

The pre-rebase hook has one or two parameters passed to it. Parameter 1 is the upstream that the current series of commits came from. Parameter 2 is the branch being rebased (or it can be empty if that branch is the current branch).

Returning a non-zero return code aborts the operation.

Post-checkout

The post-checkout hook is called whenever you successfully do a checkout in Git. It is also run after a clone unless you specify the --no-checkout option. An example of using this hook is to remove automatically created files that you don't need or want in the working directory, such as removing automatically generated backup files from an editor session.

The post-checkout hook gets three parameters. Parameter 1 is the reference of the previous HEAD (before the checkout). Parameter 2 is the reference of the current HEAD (after the checkout; this could be the same). Parameter 3 is a flag to indicate whether the checkout was for a branch or a file (1=branch and 0=file).

The exit code for this hook doesn't have any effect on the checkout because that's already been completed.

The following example script for post-checkout is written mainly as a one-line perl program wrapped in a shell script. In this example, the perl code gets a list of files, appends .bak to the filenames, and deletes those backup files if they exist.

```
#!/bin/sh
#
# Example post-checkout hook written
# Written as shell executing perl one-liner
# (c) 2016 Brent Laster

# Get a list of affected files and for each one, remove a backup file (.bak
extension) if present
/usr/bin/perl -le '@files='git ls-tree --name-only -r '$2''; chomp(@files);
@candidates=map {$_.".bak"} @files; unlink @candidates'
```

Post-merge

The post-merge hook is invoked after a successful git merge or git pull. You can use it to apply settings or data, such as permissions, after a merge is completed. (You can use a pre-commit hook to save these settings or data before the merge.) You can also use this hook to launch some process (test, install, and so on) after a merge if a particular file or files changed as a result of the merge—for example, to make sure something still builds or passes testing after being updated by a merge.

The post-merge hook gets one parameter: a flag indicating whether the merge was a squash or not. Because this hook is invoked after a merge (and only after a successful one), it cannot abort the merge or change its outcome.

Pre-push

The pre-push hook is called prior to a push to a remote. You can use it to prevent a push from happening.

Pre-push takes two parameters: the name of the destination remote (that is, origin), and the location of the destination remote (for example, `http://gitsystem.site.com/myproject`). If a named remote is not used, then parameter one also contains the location of the destination remote.

In addition to the two parameters, Git feeds this hook additional information about what is targeted for pushing through the hook's standard input (stdin). Information about each item to be pushed is passed on a separate line, and formatted as follows:

```
<local reference> <local sha1> <remote reference> <remote sha1> LF
```

Each of the SHA1 values here is the full 40-character value.

As an example, if you execute the command

```
$ git push origin master:prod
```

the hook gets an input line similar to this one:

```
refs/heads/master A1B2C3<snip>FEDF refs/heads/prod  C2EA3F<snip>DE45
```

(Here, the SHA1 values do not show a full 40-character string for brevity. Instead, I use the *<snip>* nomenclature to indicate the missing characters.)

If the remote reference doesn't exist yet, the remote SHA1 value is all zeros.

```
refs/heads/master A1B2C3<snip>FEDF refs/heads/prod 000000...0000
```

If a reference is intended to be deleted, the local reference is *(delete)* and the local SHA1 value is all zeros.

```
(delete) 000000<snip>0000 refs/heads/prod C2EA3F<snip>DE45
```

And if a non-branch reference is supplied for the local reference, it is passed as given.

```
HEAD~  A1B2C3<snip>FEDF refs/heads/prod C2EA3F<snip>DE45
```

This provides a number of options for checking what is happening with the push. If the pre-push hook returns a non-zero value, the push is aborted.

Pre-receive

The pre-receive hook is invoked on the remote side by the git receive-pack command. As you can probably tell by the name, receive-pack is one of the Git plumbing commands. Users don't normally invoke this command directly. Rather, it is invoked through a higher-level command that wraps it: git push.

Actually, git push invokes another plumbing command—git send-pack—which then invokes receive-pack. The syntax for the two plumbing commands is as follows:

```
git-receive-pack <directory>
git send-pack [--all] [--dry-run] [--force] [--receive-pack=<git-receive-pack>]
              [--verbose] [--thin] [--atomic]
              [--[no-]signed|--sign=(true|false|if-asked)]
              [<host>:]<directory> [<ref>...]
```

I won't go into detail about these plumbing commands, but you should get the idea: push needs to send data to the remote, and the remote side needs to receive it.

The pre-receive hook doesn't take any arguments. However, for each reference that is intended to be updated, the hook gets a line sent to it on stdin. Each line is of the form,

```
<old value> <new value> <reference name> LF
```

The old and new values are SHA1 values. Remember, you are updating something on the remote side (new value) from something on the local side (old value) using the push command.

You can generally think of this as one line per branch being pushed with the old and new SHA1 values for each branch in a line. A value of all zeros for one of the SHA1 values (old or new) is used to indicate a particular situation. If <old value> is equal to all zeros, that indicates a reference to be created. If <new value> is equal to all zeros, that indicates a reference to be deleted.

The pre-receive hook runs once—just before references actually start getting updated on the remote repository. If the hook exits with a non-zero return code, none of the references are updated. Even if the hook returns a zero return code, the updates of the references can still be denied by the update hook (described in the following section).

The hook sends stdout and stderr back to send-pack so the messages can be displayed to the user.

Update

The update hook is similar to the pre-receive hook. It is invoked in a similar manner through the receive-pack operation as part of a push. (See the "Pre-receive" section for more information.)

The difference between the update hook and the pre-receive hook is that the update hook is invoked once for each reference to be updated—as opposed to once for the push operation, which is the case with the pre-receive hook. As a result, you can use the update hook to allow or disallow the updating of each reference based on some checking or criteria.

The update hook takes three parameters: the name of the reference being updated, the old object's identifier (SHA1), and the new object's identifier (SHA1).

If the hook returns a non-zero return code, that reference will not be updated. Again, because this hook is called for each reference, failing one reference does not mean that all of them will be updated.

Post-receive

The post-receive hook is similar to the pre-receive hook. It is invoked in a similar manner through the receive-pack operation as part of a push. (See the "Pre-receive" section for more information.) The difference is that post-receive is invoked only after all the references have been updated. For example, you can use it to send notifications after the updates are complete or to do additional logging.

Post-receive doesn't take any arguments. However, for each reference that is intended to be updated, the hook gets a line sent to it on stdin. Each line is of the form

```
<old value> <new value> <reference name> LF
```

You can generally think of this as one line per branch being pushed with the old and new SHA1 values for each branch in a line. The values here have the same meaning as with the pre-receive hook.

The post-receive hook runs once after all the references have been updated. It has no effect on the operation or updates because all of the updates have already been done at that point. The hook sends stdout and stderr back to send-pack so that the messages can be displayed to the user.

The following example script is written in Ruby. If there is a configuration value set (user.deploy-dir), and the branch that was updated was either master or starts with *prod* or *ship*, then the hook attempts to do a *deployment* (checkout -f) of the content out to the directory specified in user .deploy-dir.

```
#!/usr/bin/env ruby

# Example post-receive hook written in ruby
# (c) 2016 Brent Laster

puts "Running post-receive hook..."
deployment_dir='git config user.deploy-dir'
# rest of hook presumes deployment_dir exists

# if the configuration value isn't set for Git, don't deploy
if (deployment_dir == "")
       puts "user.deploy_dir value not configured"
       puts "Will not deploy"
else
       # multiple lines might be passed in - one for each branch being pushed
       STDIN.each do |input_line|
             (prev_rev, new_rev, refspec) = input_line.split
             refspec.gsub!('refs/heads/','')
             # only deploy if we're on master or branch that starts with prod or ship
             if refspec =~ /^master|^(prod|ship).*$/
                    # do the deploy
                    # git-dir is already set for the hook
                    deployment_dir.chomp()
                    puts "Deploying to #{deployment_dir}"
       result = 'git --work-tree=#{deployment_dir.chomp()} checkout -f #{refspec}'
                    puts "#{result}"
             end
    end
end

exit
```

Post-update

The post-update hook is similar to the post-receive hook. It is invoked in a similar manner through the receive-pack operation as part of a push. (See the “Pre-receive” section for more information.)

Post-update takes a variable number of parameters. Each parameter is the name of a reference that was updated.

The post-update hook knows what was updated, but unlike the post-receive hook, it doesn’t know the old and new values. However, you can use it for something like updating dumb (HTTP) servers when changes are made (such as through git update-server-info). The sample hook that comes with

Git has an example of this. The hook sends stdout and stderr back to send-pack so the messages can be displayed to the user.

OTHER HOOKS

Git supports a few other hooks that are only useful in special cases. I won't go into detail about these hooks, but I will briefly mention them for completeness. You can find more information on these hooks in the git hooks documentation (git hooks --help).

Push-to-checkout

In Git, it is possible to push to a non-bare repository. A *non-bare repository* is one that has a working directory and staging area and a checked-out copy of a branch. Repositories that you push to are usually bare. They don't have a working directory or staging area because they aren't meant to have content checked out directly from them.

If you try to push to a non-bare repository, as opposed to the usual commit operation, the working directory and staging area attached to that repository won't reflect the current status, as they would if you checked things out from a local repository. So, there is a receive.denyCurrentBranch setting that can prevent this. The push-to-checkout hook can override that setting.

The push-to-checkout hook gets one parameter—the commit targeted for the update—and can return a non-zero code to block the push. Or, it can sync the working directory and the staging area up to make things consistent and return zero.

Pre-auto-gc

In the pre-auto-gc hook, *gc* refers to garbage collection—having Git clean up objects that aren't being used anymore (don't have any connections) in the repository. The *auto* part refers to an option that can be passed to that command to clean up if there are too many loose objects over a configured threshold. As a result, this hook runs first if git gc --auto is run. You can use it to do some sort of notification or verification.

The pre-auto-gc hook takes no parameters. If it returns a non-zero return code, the gc operation aborts.

Post-rewrite

The post-rewrite hook, if enabled, runs after either of two commands that rewrite history (git commit --amend or git rebase). Currently, its only parameter is the command that called it. On stdin, it receives information about what commits were rewritten in the form

```
<old sha1> <new sha1> [ <optional extra data> ]
```

In the future, additional data may be passed, but no extra data is currently passed.

HOOKS QUICK REFERENCE

Table 15-1 summarizes some of the basic information about the hooks that are available in Git. The first column contains a list of Git operations that have hooks available for them. (Note that some of these operations use only one option.) The remaining five columns identify the hooks by name and parameter descriptions for the operation, in the order of execution.

The shading of the background for each cell in the table indicates whether or not the hook can abort the operation. The key is as follows:

Black background—This hook can abort the Git operation and indicate it failed.

White background—This hook does not affect whether the operation succeeds or fails.

Gray background —This hook can abort the operation, but the hook itself may be overridden. (For the two cases here, the --no-verify option can skip running the hook.) Also, in the table, the P# values indicate a parameter passed to the hook. The value of # indicates the order it is passed in a sequence.

TABLE 15-1: List of Git Hooks by Operation

GIT OPERATION	PRE-OPERATION HOOK 1	PRE-OPERATION HOOK 2	DURING-OPERATION HOOK	POST-OPERATION HOOK 1	POST-OPERATION HOOK 2
am	**applypatch-msg** P1: Name of temporary file with proposed commit message	**pre-applypatch**		**post-apply-patch**	
commit	**pre-commit**	**prepare-commit-msg** P1: Name of temporary file with proposed commit message P2: Type of operation P3: SHA1 value for certain operations	**commit-msg** P1: Name of temporary file with proposed commit message	**post-commit**	
rebase	**pre-rebase** P1: Upstream P2: Branch being rebased (if not the same as P1)				

continues

TABLE 15-1 *(continued)*

GIT OPERATION	PRE-OPERATION HOOK 1	PRE-OPERATION HOOK 2	DURING-OPERATION HOOK	POST-OPERATION HOOK 1	POST-OPERATION HOOK 2
checkout				**post-checkout** P1: Previous HEAD (before check-out) P2: Current HEAD (after check-out) P3: Checkout type flag: 1 = branch, 2 = file	
merge, pull				**post-merge** P1: Flag that indicates squash or merge	
push	**pre-push** (local) P1: Remote reference name P2: Remote URL Extra: STDIN lines of the form <local reference> <local sha1> <remote reference> <remote sha1> LF	**pre-receive** (remote) No parameters Extra: STDIN lines of the form <old value> <new value> <reference name> LF	**update** (remote) P1: Name of reference being updated P2: Old SHA1 value P3: New SHA1 value	**post-receive** (remote) No parameters Extra: STDIN lines of the form <old value> <new value> <reference name> LF	**post-update** (remote) P* (variable number): Name of references being updated

GIT OPERATION	PRE-OPERATION HOOK 1	PRE-OPERATION HOOK 2	DURING-OPERATION HOOK	POST-OPERATION HOOK 1	POST-OPERATION HOOK 2
push (to non-bare repository)			**push-to-checkout** P1: Target commit for updating		
gc --auto			**pre-auto-gc**		
rebase, commit --amend			**post-rewrite** P1: Command that invoked it. Extra: STDIN lines of the form <old sha1> <new sha1> [<optional extra data>]		

SUMMARY

In this final chapter of the book, you've seen how to extend Git functionality through the various hooks that it provides. Each hook is a point where you can cause a script or program that you create to be executed before or after (or sometimes during) certain events related to a Git operation.

As you have seen, some of the hooks allow the user to abort the operation by returning a non-zero return code, while others are more suited for simple notifications.

Hooks can be written in any language that can be executed on the system. It is important to ensure that you understand the conditions under which the hook will be called, the arguments that it will be passed, and any environment settings that it needs to use.

Hooks can greatly enhance the usefulness users get out of Git and customize it to meet the needs of teams as well as enforce policies. For local hooks, however, it's important to establish a consistent way to ensure that all users have the same set of hooks in place on all repositories.

INDEX

A

access, 17
add command, 39–40, 52, 78–86
add subcommand (worktree), 382
aliases
  configuration, 63–64
  lab, 155–157
  parameterized, 70–71
am command, 282
amend scenario, 36
amending commits, 88–90
apply command, 281–282
apply mailbox, 282
applypatch.msg hook, 425, 427
archive command, 278–279
arguments, specifying, 53–54
auto-complete, 54
  enabling, 54–55
  Windows command prompt, 55
autocorrect, 96–97

B

Bash shell, 5, 46
binary files, 15–16, 28
  identification, 250
bisect command, 52, 290, 298–306
Bitbucket, 7
blame command, 138–140
blobs, 69
boundary commits, 138–139
branch command, 52, 162
branches, 13–14, 159–161
  checkout, 163–164
  classifications
    feature branches, 174
    integration branches, 174–175
    release and production branches, 175
    tags in branches, 175
    topic branches, 173–174
  content, 164
  creating, 162
  CVS example, 160
  deleting, 169–171
  flat files, 165
  information, 168–169
  initial development and, 172–173
  lab, 189–191
  local branches, 329
  master, 161
  private, 329
  public, 329
  remote branches, 329
    mapping, 336–338
    upstream relationships, 335–336
    viewing, 333–338
  remote tracking branches, 328–329
  renaming, 172
  testing, 162
  workflows, 175–176
    master-as-integration model, 178–179
    master-as-production model, 176
    master-as-release model, 176–178
    parallel model, 179–180
  working directory and, 164–168
  worktree command, 382
branching, 13
bundle command, 279–280
bundling, 279–280

C

cache, 37
cat-file command, 53
centralized VCS, 19, 20
check-attr command, 256
check-ignore command, 260–261

check-ins, 39
Checkout as-is
  commit as-is, 44
  commit Unix-style line endings, 44
checkout command, 40, 52, 163–164
  individual files, 186–187
Checkout Windows-style, commit Unix-style, 44
cherry command, 52
cherry-pick command, 52
cherry-picking, 203–205
  rebase comparison, 205–206
clean command, 286–288
clean filter, 251–254
cleanup commands, 285–290
clone command, 40, 52, 65–66, 330–333
  bare option, 333
  branch option, 333
  clone paths, 332
  depth option, 333
  mirror option, 333
  single-branch option, 333
cloning, 20
  clone command, 40, 65–66, 330–332
    bare option, 333
    branch option, 333
    clone paths, 332
    depth option, 333
    mirror option, 333
    single-branch option, 333
  lab, 353–354
  submodules and, 390–393
code, shared, 29
collaboration models, hosted repositories, 368–369
  fork-and-pull, 369–372
  local updates, 373–374
  multiple remotes, 372–373
collection points, 322
command line interface, 5
commands, 52
  add, 39, 40, 52
  aliases, 63
  am, 282
  apply, 281–282
  archive, 278–279
  arguments, 53–54
  auto-complete and, 54–55
  bisect, 52, 290, 298–306
  blame, 138–140
  branch, 52, 162
  bundle, 279–280
  cat-file, 53
  check-attr, 256
  check-ignore, 260–261
  checkout, 40, 52, 163–164
  cherry, 52
  cherry-pick, 52
  clean, 286–288
  cleanup commands, 285–290
  clone, 40, 52, 65–66, 330–333
    bare option, 333
    branch option, 333
    clone paths, 332
    depth option, 333
    mirror option, 333
    single-branch option, 333
  commit, 39–40, 52, 86–87
    edit skipping, 90
    full commit, 95–96
    prerequisites, 87
    scope, 88
    verbose option, 95
  commit-tree, 53
  components, 50
  config, 52
  content moving, 39–41
  count-objects, 53
  diff, 52, 111, 113–114, 121–124
    alternative forms, 120
    difftool, 121–124
    filemode changes, 118
    names only, 115–116
    SHA1, 115
    specific version, 115
    staging area, 114–115
    symbolic names, 112
    two commits, 118–120
    visual diffing, 121–124
    whitespace changes, 116–118
    word-diff, 116
  diff-index, 53
  executing, 49–55
  fetch, 40, 52, 346–347
    branch synchronizing, 348–349
    depth option, 348
    force option, 348
  filter-branch, 290–291
    subdirectory split to repository, 292–294
    undoing a filter-branch, 294
  for-each-ref, 53

form, 49–50
gc, 288
grep, 52
hash-object, 53
help, 52, 73–75
log, 52, 132–136, 275–278
    display options, 132–133
    filtering options, 132–133
    history, 134–137
    output, 135–136
    time limiting options, 133–134
ls-files, 53
merge, 52, 194
merge-base, 53
merges, pull command options, 350–351
mv, 52, 271
notes, 288–290
operands, types, 50–52
plumbing, 52
    cat-file, 53
    commit-tree, 53
    count-objects, 53
    diff-index, 53
    for-each-ref, 53
    hash-object, 53
    ls-files, 53
    merge-base, 53
    read-tree, 53
    rev-list, 53
    rev-parse, 53
    show-ref, 53
    symbolic-ref, 53
    update-index, 53
    update-ref, 53
    verify-pack, 53
    write-tree, 53
porcelain, 52
    add, 52
    bisect, 52
    branch, 52
    checkout, 52
    cherry, 52
    cherry-pick, 52
    clone, 52
    commit, 52
    config, 52
    diff, 52
    fetch, 52
    grep, 52
    help, 52
    log, 52
    merge, 52
    mv, 52
    pull, 52
    push, 52
    rebase, 52
    rerere, 52
    reset, 52
    revert, 52
    rm, 52
    show, 52
    status, 52
    submodule, 52
    subtree, 52
    tag, 52
    worktree, 52
pull, 40–41, 52, 349–350
    fetching options, 351–352
    merging options, 350–351
    recurse-submodules option, 395
push, 39–40, 52, 338–340
    all option, 344
    defaults, 343
    delete option, 344
    follow-tags option, 344
    force option, 344–346
    formats, 342–343
    mirror option, 346
    set-upstream option, 346
    status, 340–342
    tags option, 344, 346
read-tree, 53
rebase, 52
rebase master, 201
rerere, 52, 290, 306–314
reset, 52
revert, 52
rev-list, 53, 291–292
rev-parse, 53
rm, 52, 272
searching and, 272–278
send-email, 283–285
show, 52
show-ref, 53
staging, 270
stash, 194–195, 264–270
    apply option, 269–270
    description, 266–267
    options, 265
    pop option, 269–270

restoring changes, 268
syntax, 264
uncommitted changes, 264
untracked files and, 265–266
viewing stash, 267–268
status, 52, 105–111
short form, 110–111
tracked files, 106
workflow and, 106–110
submodule, 52, 388–389
add operation, 388–389
foreach subcommand, 393–395
recurse-submodules option, 402–404
status subcommand, 389–390, 396–397
update subcommand, 395, 398
submodules, init sub command, 391
subtree, 52, 405–406
push subcommand, 411
split subcommand, 410
subtree merge, 409
symbolic-ref, 53
tag, 52
update-index, 53
update-ref, 53
verify-pack, 53
worktree, 52, 382
branches, 382
subcommands, 382, 385–386
worktree command, 382
write-tree, 53
commit command, 39–40, 52, 86–87
edit skipping, 90
full commit, 95–96
prerequisites, 87
scope, 88
verbose option, 95
commit-msg hook, 424, 425, 431–432
commits, 34, 69
autocorrect, 96–97
boundary commit, 138–139
commit command, 39, 40, 86–87
edit skipping, 90
full commit, 95–96
prerequisites, 87
scope, 88
verbose option, 95
e-mail, changing, 297–293
hooks and, 439, 441
merge commits, 197, 199
non-branch, 181–186
ranges, 204–205
repairing, 36
results, 91–94
saving uncommitted changes, 264
signing, 151–152
tags, 141–142, 152–153
details, 142
example, 144
modifying, 143
commit-tree command, 53
config command, 52
configuration, 55–56
aliases, 63–64
EOL (end of line), 61–63
files, 5
global level, 57
hierarchy, 58
listing settings, 59–60
local level, 58
one-shot configuration, 60
system level, 57
undoing settings, 59
user information, 56–57
values, viewing, 58–59
conflicts, remote repository, 356–368
merges after pull, 361–364
pull alone, 364–365
pull with rebase, 365–367
push updated content, 367–368
resolution, 361
scope, 357–360
content
adding, 77–78
staging scope, 78–86
branches, 164
building, 33
moving, 271–272
commands, 39–41
to local environment, 41
to local repository, 40
to remote repository, 40
to staging area, 40
to working directory, 41

remote repository, 33
renaming, 271
count-objects command, 53
CVCS (centralized version control system), 19, 20

D

daemon program, remote access and, 324
Debian, package manager, 6
default location override, 34
deleting, lab, 317–319
delta storage, 22–23
deltas, rebasing and, 199
dev-test-prod environment, 31–33
diff command, 52, 111, 113–114, 121–124
alternative forms, 120
difftool, 121–124
filemode changes, 118
names only, 115–116
SHA1, 115
specific version, 115
staging area, 114–115
symbolic names, 112
two commits, 118–120
visual diffing, 121–124
whitespace changes, 116–118
word-diff, 116
diff-index command, 53
directories
hooks, 424
trees, connecting to, 34
working directory, 33–34
disconnected development, 21
distributed VCS, 19, 20–21
DVCS (distributed version control system), 19, 20–21

E

ease-of-use packages, 8–9
Easy Git, 5
editor, default, 60–61
EGit, 5
Enable file system, 45
environment variables, hooks and, 427
EOL (end of line), 61–63
error messages, rebasing, 211–212
executables, 5
executing commands, 49–55
extensions, 27–28
disabling hooks, 424

F

fast-forward merges, 195–196
feature branches, 174
Fedora, package manager, 6
fetch
fetch command, 40, 346–347
branches, synchronizing, 348–349
depth option, 348
force option, 348
pull command options
depth, 351
force, 351–352
fetch command, 40, 52, 346–347
branch synchronizing, 348–349
depth option, 348
force option, 348
file formats
Git attributes file, 248–249
Git ignore file, 258
overrides, 259
pattern negation, 258
targeted inclusion, 259–260
files
deleting from all history, 294–297
ignoring in merges, 254–255
lifecycle, lab, 127–129
pack files, 23
patch files, mailing, 283–285
storage model, 27
untracked, stash command and, 265–266
filter-branch command, 290–291
subdirectory split to repository, 292–294
undoing a filter-branch, 294
filters
drivers, 251
Git attributes file
clean, 251–254
smudge, 251–254
flat files, branches, 165
for-each-ref command, 53
fork and pull model, 6
FreeBSD, package manager, 6

G

gc command, 288
Gentoo, package manager, 6
Gerrit, 10
Git
  companies using, 5
  content, moving, 14
  core Git, 5–6
  disconnected development, 11
  ease of use, 12
  history, 4
  introduction, 3–4
  performance, 11–12
  projects, 5
  SHA1, 12
  staging area, 13
  version rewriting, 12–13
Git attributes file, 245–246
  attributes
    options, 249
    scope, 246–248
  binary files, identification, 250
  check-attr command, 256
  col attribute, 250
  diff attribute, 255
  file format, 248–249
  files, ignoring in merges, 254–255
  filters
    clean, 251–254
    drivers, 251
    smudge, 251–254
  ident attribute, 255
  line endings, 250–251
  paths, ignoring in merges, 254–255
  text attribute, 250
  use cases, 249–256
git config command, mapping to configuration files, 69–70
Git Credential Manager for Windows, 45
Git Extensions, 5, 8
Git for Windows, 5
git gui, 5
Git ignore file, 245–246, 256–257
  check-ignore command, 260–261
  file format, 258
    overrides, 259
    pattern negation, 258
    targeted inclusion, 259–260
  scope, 257–258
git init command, 64–65
  running twice on repository, 67
.git subdirectory, 34
GIT-DIR, hooks and, 427
--git-dir option, 34
Git-hosting sites, 6–7
GitHub, 5
  account setup, lab, 353–354
  example page, 7
.gitignore file, 34
gitk, 5
GitLab, 7–8
.gitmodules file, 387
Gitolite, 5
grep command, 52, 272–275
GUIs, 8–9

H

hash-object command, 53
help, 73–75
  OS X, 75
help command, 52, 73–75
history
  changes, canceling, 147–150
  files, deleting from all history, 294–297
  lab, 155–157
  local environment, reset, 146
  log command and, 134–135
  rewriting, 36
  rollbacks, 145–146
  searching, 136–137
  visual view, 140–141
Homebrew, 5
hooks, 423
  am operation, 439
  applypatch.msg, 425, 427
  checkout and, 440
  commit-msg, 425, 431–432
  commits, 439, 441
  environment variables and, 427
  extensions disabling, 424
  gc --auto and, 440
  installation, 424
  merges and, 440
  post-applypatch, 425, 428
  post-checkout, 425, 434

post-commit, 425, 432–433
post-merge, 426, 434
post-rewrite, 426, 438
pre-applypatch, 425, 427–428
pre-commit, 425, 428–429
prepare-commit-msg, 425, 429–431
pre-push, 426, 434–435
pre-rebase, 425, 433–434
pull and, 440
push and, 440
push-to-checkout, 426, 438
rebase and, 439, 441
remote
post-receive, 426, 436–437
post-update, 426, 437
pre-aut-gc, 426, 438
pre-receive, 426, 435–436
update, 426, 436
return code and, 426
updating, 425
uses, 423
working directory and, 426–427
hooks directory, 424
hosted repositories, 368–369
collaboration models
fork-and-pull, 369–372
local updates, 373–374
multiple remotes, 372–373
HTTP, 325–326

I

index, 37
index-filter option, 294–297
industry standards, 4–5
installation
Git for Windows, 43–46
Linux, 6
on Linux, 4–
on Mac OS X, 46
integration branches, 174–175
interactive rebasing, 193

J

JAR files, 34
JARs, 25
JGit, 10

L

labs
aliases, 155–157
branches, 189–191
cloning, 353–354
deleting, 317–319
file lifecycle, 127–129
GitHub account setup, 353–354
history, 155–157
merges, 243–244
mv command, 317–319
remote repository, 377–380
repositories, 99–103
repository cloning, 353–354
rm command, 317–319
submodules, 415–417
subtrees, 419–421
tags, 155–157
worktrees, 413–414
levels, 31–32
dev-test-prod environment, 31–33
staging area, 32
LFS (Large File Storage), 27
libraries, JGit, 10
linked working tree, 382
Linux
Git installation, 46
package managers, 6
list subcommand (worktree), 385
local directory tree, connection, 34
local environment, 20, 33
resetting, 146
Local protocol, 324
local repository, 21, 37–38
local environment and, 33
log command, 52, 132–136, 275–278
display options, 132–133
filtering options, 132–133
history, 134–137
searching, 136–137
output, 135–136
time limiting options, 133–134
ls-files command, 53

M

Mac OS X
Git installation, 46

web-based help, 75
MacPorts, 5
main working tree, 382
master branch, 107
merges and, 195–196
merge command, 52, 194
merge commits, 197, 199
merge-base command, 53
merges
aborting, 212
cherry-picking, 203–205
conflicts
alternative style, 227–229
error messages, 211–212
merge as state, 210–211
workflow, 213–217
destination branch, 195
fast-forward, 195–196
files, ignoring, 254–255
hooks and, 440
lab, 243–244
merge command, 194
octopus strategy, 221–222
our strategy, 222
paths, ignoring, 254–255
preparations, 194–195
pull command options
no-commit, 350
no-edit, 350
rebase, 350–351
strategy, 351
strategy-option, 351
rebasing
deltas, 199
replays, 199
recursive strategy, 217–221
resolve strategy, 217
SHA1 value
logs, 210
ORIG_HEAD and, 207–208
reflogs, 208–210
tags, 210
source branch, 195
staging area and, 37, 215–216
stash command, 269–270
as state, 210–211
submodules and, 404
subtree merge command, 409
three-way, 196–199
undoing, 206–207
visual merging, 223–225
tool availability, 226–227
tool selection, 226
moving content, mv command, 271–272
mv command, 52, 271
lab, 317–319

N

non-branch commits, 181–186
notes command, 288–290

O

octopus strategy, merges, 221–222
OpenBSD, package manager, 6
operands, types, 50–52
OS/X, 5
our strategy, merges, 222
output, log command, 135–136

P

pack files, 23
package managers, Linux systems, 6
parameterized aliases, 70–71
patch sets, 22
patches
am command, 282
apply command, 281–282
mailing patch files, 283–285
sharing, e-mail and, 280–281
paths, merges, ignoring, 254–255
permissions, 17
pickaxe option, 276
Plink (PuTTy linke), 44
plug-ins, 10
plumbing commands, 51–53
cat-file, 53
commit-tree, 53
count-objects, 53
diff-index, 53
filter-branch, 291
for-each-ref, 53
hash-object, 53
ls-files, 53
merge-base, 53
read-tree, 53

rev-list, 53, 291–292
rev-parse, 53
show-ref, 53
symbolic-ref, 53
update-index, 53
update-ref, 53
verify-pack, 53
write-tree, 53
porcelain commands, 51–53
add, 52
bisect, 52
branch, 52
checkout, 52
cherry, 52
cherry-pick, 52
clone, 52
commit, 52
config, 52
diff, 52
fetch, 52
grep, 52
help, 52
log, 52
merge, 52
mv, 52
pull, 52
push, 52
rebase, 52
rerere, 52
reset, 52
revert, 52
rm, 52
show, 52
status, 52
submodule, 52
subtree, 52
tag, 52
worktree, 52
post-applypatch hook, 425, 428
post-checkout hook, 425, 434
post-commit hook, 425, 432–433
post-merge hook, 426, 434
post-receive remote hook, 426, 436–437
post-rewrite hook, 426
post-update remote hook, 426, 437
pre-applypatch hook, 425, 427–428
pre-aut-gc remote hook, 426, 438
pre-commit hook, 425, 428–429
Prepare scenario, 35–36
prepare-commit-msg hook, 425, 429–431
pre-push hook, 426, 434–435
pre-rebase hook, 425, 433–434
pre-receive remote hook, 426, 435–436
projects, as subtrees, 407–409
promotion model example, 32–33
prune subcommand (worktree), 385–386
pull command, 40, 41, 52, 349–350
fetching options
depth, 351
force, 351–352
merging options
no-commit, 350
no-edit, 350
rebase, 350–351
strategy, 351
strategy-option, 351
recurse-submodules option, 395
pull commands, 40–41, 52, 349–350
fetching options, 351–352
merging options, 350–351
recurse-submodules option, 395
push command, 39–40, 52, 338–340
all option, 344
defaults, 343
delete option, 344
follow-tags option, 344
force option, 344–346
formats, 342–343
mirror option, 346
set-upstream option, 346
status, 340–342
tags option, 344, 346
push-to-checkout hook, 426

R

read-tree command, 53
rebase command, 52
rebase master command, 201
rebasing
advanced scenario, 229–233
cherry-picking comparison, 205–206
deltas, 199
error messages, 211–212
interactive
action selection, 237
commit editing, 241

drop action, 235
edit action, 235
example, 236
exec action, 235
file format, 237
fixup action, 235
pick action, 235
preparation, 234
reword action, 235
script running, 237–241
squash action, 235
workflow, 234–235
replays, 199
syntax, 200–201
recursive strategy, merges, 217–221
reflogs, 153–154
release and production branches, 175
remote access
daemon program, 324
HTTP, 325–326
Local protocol, 324
remote command, 326–328
SSH, 324–325
remote branches
mapping, 336–338
upstream relationships, 335–336f
viewing, 333–338
remote command, 326–328
remote environment, 33
interaction, 328
remote tracking branches, 328–329
remote hooks
post-receive, 426, 436–437
post-update, 426, 437
pre-aut-gc, 426, 438
pre-receive, 426, 435–436
update, 426, 436
remote references, 322, 327
remote repository, 20–21, 33, 38–39, 322
bundle command, 280
conflicts, 356–357
merges after pull, 361–364
pull alone, 364–365
pull with rebase, 365–367
push updated content, 367–368
resolution, 361
scope, 357–360
lab, 377–380
references, 322
remotes, 322–324
submodule updates, 394–396
renaming content, 271
Repair scenario, 36
repositories. *See also* local repository; remote repository
branches, 39
cloning, lab, 353–354
design
file scope, 27–29
repository scope, 25–27
fork and pull model, 6
hosted, 368–369
collaboration models, 369–374
initialization
clone command, 65–66
git init command, 64–65
lab, 99–103
local repository, 21
management, 5
multiple, 76–77
remote, 20–21, 322
scope, 25–27
separate repositories, 27
skeleton, 34
staging area becomes, 33
as subdirectory, submodules and, 387
rerere command, 52, 290, 306–314
reset command, 52
resolve strategy, merges, 217
revert, 147–150
revert command, 52
rev-list command, 53, 291–292
rev-parse command, 53
rewrite, 36
rm command, 52, 272
lab, 317–319
rollbacks, 145–146
root directory, 46
rr-cache, 309–310

S

SCM (source code management), 4
scope, 16
searches
commands
grep, 272–275
log, 275–278

history, 136–137
self-hosting packages, 7–8
send-email command, 283–285
SHA1, 12, 67–68
show command, 52
show-ref command, 53
SmartGit, 8
smudge filter, 251–254
snapshot storage, 23–24
Solaris 11 Express, package manager, 6
source management, 3
SourtTree, 8
SSH (Secure Shell), 324–325
staging area, 13, 14–15
amend scenario, 36
bypassing, 37
cache, 37
index, 37
local environment and, 33
merging and, 37, 215–216
Prepare scenario, 35–36
purpose, 35
Repair scenario, 36
when to use, 36–37
staging commands, 270
staging scope
bypassing area, 86
commits interactive staging, 80–81
editing hunks, 80–81
ignoring files, 78
partial staging, 78–81
stash command, 194–195, 264–270
apply option, 269–270
merges and, 269–270
description, 266–267
options, 265
pop option, 269–270
merges and, 269–270
restoring changes, 268
syntax, 264
uncommitted changes, 264
untracked files and, 265–266
viewing stash, 267–268
status command, 52, 105–111
short form, 110–111
tracked files, 106
workflow and, 106–110
storage
delta storage, 22–23
files, 27
requirements, 23–25
snapshot storage, 23–24
submodule command, 52, 388–389
add operation, 388–389
foreach subcommand, 393–395
recurse-submodules option, 402–404
status subcommand, 389–390, 396–397
update subcommand, 395, 398
submodules, 33
adding, 388–389
changes, pushing, 402–404
differences, viewing, 396–397
init sub command, 391
lab, 415–417
merges and, 404
populating, 390–392
problems, 399
processing multiple, 393–394
references, updating, 399–401
repository as subdirectory, 387
status determination, 389–390
cloning and, 390–393
submodule command
add operation, 388–389
foreach subcommand, 393–395
recurse-submodules option, 402–404
status subcommand, 389–390, 396–397
update subcommand, 395, 398
superproject comparison, 397–399
syntax, 386
unregistering, 404–405
updating, when superproject is updated, 401–402
updating from remotes, 394–396
subprojects, 138–139
subtree command, 52, 405–406
push subcommand, 411
split subcommand, 410
subtree merge command, 409
subtrees, 405–406
lab, 419–421
projects as, 407–409
splits, 410
new projects from, 410–411
updating, 409
symbolic names, 112
symbolic-ref command, 53

T

tag command, 52
tags, 141–142
  in branches, 175
  details, 142
  example, 144
  lab, 155–157
  modifying, 143
  signing, 151–153
testing
  branches, 162
  checkout testing, 163
three-way merges, 196–199
timestamps, 17
topic branches, 173–174
TortoiseGit, 8
tracked files, 106
tree-filter option, 294–297
trees, 69
triggers, 423

U

Ubuntu, package manager, 6
uncommitted changes,
    saving, 264
update remote hook, 426, 436
update-index command, 53
update-ref command, 53
updating hook, 425
Use MinTTY option, 45

V

VCS (version control systems), 19
  centralized, 19
  distributed, 19
verify-pack command, 53
version numbers, 16
visual diffing, 121–124
visual merging, 223–225
  tools
    availability, 226–227
    selecting, 226

W–Z

Windows, 5. *See also* Git for Windows
  filesystem cache, 64
  Git installation, 43–46
word-diff, 116
workflows, 356–367
  branches, 175–176
    master-as-integration model, 178–179
    master-as-production model, 176
    master-as-release model, 176–178
    parallel model, 179–180
  hosted repositories, 368–369
    collaboration models
      fork-and-pull, 369–372
      local updates, 373–374
      multiple remotes, 372–373
  merge conflicts, 213–217
  remote repository, conflicts, 356–368
  return code and, 426
  status command and, 106–110
working directory, 33–34
  branches and, 164–168
  hooks and, 426–427
  local environment and, 33
workspaces, 33–34
worktree command, 52, 382
  branches, 382
  subcommands, 382, 385–386
worktrees, 381
  adding, 382–384
  lab, 413–414
  linked working tree, 382
  main working tree, 382
  pruning, 385–386
  worktree command, 382
    branches, creating, 382
    subcommands, 382, 385–386
write-tree command, 53

WILEY END USER LICENSE AGREEMENT

Go to www.wiley.com/go/eula to access Wiley’s ebook EULA.