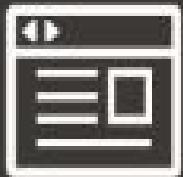


Ferdinando Santacroce, Aske Olsson
Rasmus Voss, Jakub Narębski

Git: Mastering Version Control

Learn everything you need to take full control of your workflow with Git with this curated Learning Path - dive in and transform the way you work



Packt

Git: Mastering Version Control

Table of Contents

[Git: Mastering Version Control](#)

[Git: Mastering Version Control](#)

[Credits](#)

[Preface](#)

[What this learning path covers](#)

[What you need for this learning path](#)

[Who this learning path is for](#)

[Reader feedback](#)

[Customer support](#)

[Downloading the example code](#)

[Errata](#)

[Piracy](#)

[Questions](#)

[I. Module 1](#)

[1. Getting Started with Git](#)

[Installing Git](#)

[Running our first Git command](#)

[Setting up a new repository](#)

[Adding a file](#)

[Commit the added file](#)

[Modify a committed file](#)

[Summary](#)

[2. Git Fundamentals – Working Locally](#)

[Repository structure and file status life cycle](#)

[The working directory](#)

[File statuses](#)

[The staging area](#)

[Unstaging a file](#)

[The time metaphor](#)

[The past](#)

[The present](#)

[The future](#)

Working with repositories

Unstaging a file

Viewing the history

Anatomy of a commit

The commit snapshot

The commit hash

Author, e-mail, and date

Commit messages

Committing a bunch of files

Ignoring some files and folders by default

Highlighting an important commit – Git tags

Taking another way – Git branching

Anatomy of branches

Looking at the current branches

Creating a new branch

Switching from branch to branch

Understanding what happens under the hood

A bird's eye view to branches

Typing is boring – Git aliases

Merging branches

Merge is not the end of the branch

Exercises

Exercise 2.1

What you will learn

Scenario

Results

Exercise 2.2

What you will learn

Scenario

Results

Deal with branches' modifications

Difffing branches

Using a visual diff tool

Resolving merge conflicts

Edit collisions

Resolving a removed file conflict

Keeping the edited file
Resolving conflicts by removing the file

Stashing

Summary

3. Git Fundamentals – Working Remotely

Working with remotes

Setting up a new GitHub account

Cloning a repository

Uploading modifications to remotes

What do I send to the remote when I push?

Pushing a new branch to the remote

The origin

Tracking branches

Downloading remote changes

Checking for modifications and downloading them

Applying downloaded changes

Going backward: publish a local repository to GitHub

Adding a remote to a local repository

Pushing a local branch to a remote repository

Social coding – collaborate using GitHub

Forking a repository

Submitting pull requests

Creating a pull request

Summary

4. Git Fundamentals – Niche Concepts, Configurations, and Commands

Dissecting the Git configuration

Configuration architecture

Configuration levels

System level

Global level

Repository level

Listing configurations

Editing configuration files manually

Setting up other environment configurations

Basic configurations

[Typos autocorrection](#)
[Push default](#)
[Defining the default editor](#)
[Other configurations](#)

[Git aliases](#)

[Shortcuts to common commands](#)

[Creating commands](#)

[git unstage](#)

[git undo](#)

[git last](#)

[git difflast](#)

[Advanced aliases with external commands](#)

[Removing an alias](#)

[Aliasing the git command itself](#)

[Git references](#)

[Symbolic references](#)

[Ancestry references](#)

[The first parent](#)

[The second parent](#)

[World-wide techniques](#)

[Changing the last commit message](#)

[Tracing changes in a file](#)

[Cherry picking](#)

[Tricks](#)

[Bare repositories](#)

[Converting a regular repository to a bare one](#)

[Backup repositories](#)

[Archiving the repository](#)

[Bundling the repository](#)

[Summary](#)

[5. Obtaining the Most – Good Commits and Workflows](#)

[The art of committing](#)

[Building the right commit](#)

[Make only one change per commit](#)

[Split up features and tasks](#)

[Write commit messages before starting to code](#)

Include the whole change in one commit

Describe the change, not what you have done

Don't be afraid to commit

Isolate meaningless commits

The perfect commit message

Writing a meaningful subject

Adding bulleted details lines, when needed

Tie other useful information

Special messages for releases

Conclusions

Adopting a workflow – a wise act

Centralized workflows

How they work

Feature branch workflow

GitFlow

The master branch

Hotfixes branches

The develop branch

The release branch

The feature branches

Conclusion

The GitHub flow

Anything in the master branch is deployable

Creating descriptive branches off of the master

Pushing to named branches constantly

Opening a pull request at any time

Merging only after a pull request review

Deploying immediately after review

Conclusion

Other workflows

The Linux kernel workflow

Summary

6. Migrating to Git

Before starting

Prerequisites

Working on a Subversion repository using Git

[Creating a local Subversion repository](#)
[Checking out the Subversion repository with svn client](#)
[Cloning a Subversion repository from Git](#)
 [Setting up a local Subversion server](#)
 [Adding a tag and a branch](#)
 [Committing a file to Subversion using Git as a client](#)
[Using Git with a Subversion repository](#)
[Migrating a Subversion repository](#)
 [Retrieving the list of Subversion users](#)
 [Cloning the Subversion repository](#)
 [Preserving the ignored file list](#)
 [Pushing to a local bare Git repository](#)
 [Arranging branches and tags](#)
 [Renaming the trunk branch to master](#)
 [Converting Subversion tags to Git tags](#)
 [Pushing the local repository to a remote](#)
[Comparing Git and Subversion commands](#)
[Summary](#)

[7. Git Resources](#)

[Git GUI clients](#)

[Windows](#)

[Git GUI](#)

[TortoiseGit](#)

[GitHub for Windows](#)

[Atlassian SourceTree](#)

[Cmder](#)

[Mac OS X](#)

[Linux](#)

[Building up a personal Git server with web interface](#)

[The SCM Manager](#)

[Learning Git in a visual manner](#)

[Git on the Internet](#)

[Git community on Google+](#)

[GitMinutes and Thomas Ferris Nicolaisen's blog](#)

[Ferdinando Santacroce's blog](#)

[Summary](#)

II. Module 2

1. Navigating Git

Introduction

Git's objects

Getting ready

How to do it...

The commit object

The tree object

The blob object

The branch

The tag object

How it works...

There's more...

See also

The three stages

Getting ready

How to do it...

How it works...

See also

Viewing the DAG

Getting ready

How to do it...

How it works...

See also

Extracting fixed issues

Getting ready

How to do it...

How it works...

There's more...

Getting a list of the changed files

Getting ready

How to do it...

How it works...

There's more...

See also

Viewing history with Githk

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[Finding commits in history](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[Searching through history code](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[2. Configuration](#)

[Configuration targets](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[Querying the existing configuration](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[Templates](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[A .git directory template](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[A few configuration examples](#)

[Getting ready](#)

How to do it...

Rebase and merge setup

Expiry of objects

Autocorrect

How it works...

There's more...

Git aliases

Getting ready

How to do it...

How it works...

There's more...

The refspec exemplified

Getting ready

How to do it...

How it works...

3. Branching, Merging, and Options

Introduction

Managing your local branches

Getting ready

How to do it...

How it works...

There's more...

Branches with remotes

Getting ready

How to do it...

There's more...

Forcing a merge commit

Getting ready

How to do it...

There's more...

Using git rerere to merge known conflicts

How to do it...

There's more...

The difference between branches

Getting ready

How to do it...

[There's more...](#)

[4. Rebase Regularly and Interactively, and Other Use Cases](#)

[Introduction](#)

[Rebasing commits to another branch](#)

[Getting ready](#)

[How to do it...](#)

[How it works](#)

[Continuing a rebase with merge conflicts](#)

[How to do it](#)

[How it works](#)

[There's more...](#)

[Rebasing selective commits interactively](#)

[Getting ready](#)

[How to do it](#)

[There's more...](#)

[Squashing commits using an interactive rebase](#)

[Getting ready](#)

[How to do it...](#)

[There's more...](#)

[Changing the author of commits using a rebase](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[Auto-squashing commits](#)

[Getting ready](#)

[How to do it...](#)

[There's more...](#)

[5. Storing Additional Information in Your Repository](#)

[Introduction](#)

[Adding your first Git note](#)

[Getting ready](#)

[How to do it...](#)

[There's more...](#)

[Separating notes by category](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[Retrieving notes from the remote repository](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[Pushing notes to a remote repository](#)

[How to do it...](#)

[There's more...](#)

[Tagging commits in the repository](#)

[Getting ready](#)

[How to do it...](#)

[There's more...](#)

[6. Extracting Data from the Repository](#)

[Introduction](#)

[Extracting the top contributor](#)

[Getting ready](#)

[How to do it...](#)

[There's more...](#)

[Finding bottlenecks in the source tree](#)

[Getting ready](#)

[How to do it...](#)

[There's more...](#)

[Grepping the commit messages](#)

[Getting ready](#)

[How to do it...](#)

[The contents of the releases](#)

[How to do it...](#)

[How it works...](#)

[7. Enhancing Your Daily Work with Git Hooks, Aliases, and Scripts](#)

[Introduction](#)

[Using a branch description in the commit message](#)

[Getting ready](#)

[How to do it...](#)

[Creating a dynamic commit message template](#)

[Getting ready](#)

[How to do it...](#)

[There's more...](#)

[Using external information in the commit message](#)

[Getting ready](#)

[How to do it...](#)

[Preventing the push of specific commits](#)

[Getting ready](#)

[How to do it...](#)

[There's more...](#)

[Configuring and using Git aliases](#)

[How to do it...](#)

[How it works...](#)

[Configuring and using Git scripts](#)

[How to do it...](#)

[Setting up and using a commit template](#)

[Getting ready](#)

[How to do it...](#)

[8. Recovering from Mistakes](#)

[Introduction](#)

[Undo – remove a commit completely](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[Undo – remove a commit and retain the changes to files](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[Undo – remove a commit and retain the changes in the staging area](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[Undo – working with a dirty area](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[Redo – recreate the latest commit with new changes](#)

[Getting ready](#)
[How to do it...](#)
[How it works...](#)
[There is more...](#)

[Revert – undo the changes introduced by a commit](#)

[Getting ready](#)
[How to do it...](#)
[How it works...](#)
[There's more...](#)

[Reverting a merge](#)

[Getting ready](#)
[How to do it...](#)
[How it works...](#)
[There is more...](#)
[See also](#)

[Viewing past Git actions with git reflog](#)

[Getting ready](#)
[How to do it...](#)
[How it works...](#)

[Finding lost changes with git fsck](#)

[Getting ready](#)
[How to do it...](#)
[How it works...](#)
[See also](#)

[9. Repository Maintenance](#)

[Introduction](#) [Pruning remote branches](#)

[Getting ready](#)
[How to do it...](#)
[How it works...](#)
[There's more...](#)

[Running garbage collection manually](#)

[Getting ready](#)
[How to do it...](#)
[How it works...](#)

[Turning off automatic garbage collection](#)

[Getting ready](#)

[How to do it...](#)

[Splitting a repository](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[Rewriting history – changing a single file](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[Back up your repositories as mirror repositories](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[A quick submodule how-to](#)

[Getting ready](#)

[How to do it...](#)

[There's more...](#)

[Subtree merging](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[Submodule versus subtree merging](#)

[10. Patching and Offline Sharing](#)

[Introduction](#)

[Creating patches](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[Creating patches from branches](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[Applying patches](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[Sending patches](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[Creating Git bundles](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[Using a Git bundle](#)

[Getting ready](#)

[How to do it...](#)

[There's more...](#)

[Creating archives from a tree](#)

[Getting ready](#)

[How to do it...](#)

[There's more...](#)

[11. Git Plumbing and Attributes](#)

[Introduction](#)

[Displaying the repository information](#)

[Getting ready](#)

[How to do it...](#)

[There's more...](#)

[Displaying the tree information](#)

[Getting ready](#)

[How to do it...](#)

[Displaying the file information](#)

[Getting ready](#)

[How to do it...](#)

[There's more...](#)

[Writing a blob object to the database](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[Writing a tree object to the database](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[Writing a commit object to the database](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[Keyword expansion with attribute filters](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[Metadata diff of binary files](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[Storing binaries elsewhere](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Checking the attributes of a file](#)

[Getting ready](#)

[How to do it...](#)

[Attributes to export an archive](#)

[Getting ready](#)

[How to do it...](#)

[There's more...](#)

[12. Tips and Tricks](#)

[Introduction](#)

[Using git stash](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[Saving and applying stashes](#)

[Getting ready](#)

[How to do it...](#)

[There's more...](#)

[Debugging with git bisect](#)

[Getting ready](#)

[How to do it...](#)

[There's more...](#)

[Using the blame command](#)

[Getting ready](#)

[How to do it...](#)

[There's more...](#)

[Color UI in the prompt](#)

[Getting ready](#)

[How to do it...](#)

[There's more...](#)

[Autocompletion](#)

[Getting ready](#)

[Linux](#)

[Mac](#)

[Windows](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[Bash prompt with status information](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[More aliases](#)

[Getting ready](#)

[How to do it...](#)

[Interactive add](#)

[Getting ready](#)

[How to do it...](#)

[There's more...](#)

[Interactive add with Git GUI](#)

[Getting ready](#)

[How to do it...](#)

[Ignoring files](#)

[Getting ready](#)

[How to do it...](#)

[There's more...](#)

[See also...](#)

[Showing and cleaning ignored files](#)

[Getting ready](#)

[How to do it...](#)

[There's more...](#)

III. Module 3

1. Git Basics in Practice

[An introduction to version control and Git](#)

[Git by example](#)

[Repository setup](#)

[Creating a Git repository](#)

[Cloning the repository and creating the first commit](#)

[Publishing changes](#)

[Examining history and viewing changes](#)

[Renaming and moving files](#)

[Updating your repository \(with merge\)](#)

[Creating a tag](#)

[Resolving a merge conflict](#)

[Adding files in bulk and removing files](#)

[Undoing changes to a file](#)

[Creating a new branch](#)
[Merging a branch \(no conflicts\)](#)
[Undoing an unpublished merge](#)

[Summary](#)

[2. Exploring Project History](#)

[Directed Acyclic Graphs](#)

[Whole-tree commits](#)

[Branches and tags](#)

[Branch points](#)

[Merge commits](#)

[Single revision selection](#)

[HEAD – the implicit revision](#)

[Branch and tag references](#)

[SHA-1 and the shortened SHA-1 identifier](#)

[Ancestry references](#)

[Reverse ancestry references: the git describe output](#)

[Refllog shortnames](#)

[Upstream of remote-tracking branches](#)

[Selecting revision by the commit message](#)

[Selecting the revision range](#)

[Single revision as a revision range](#)

[Double dot notation](#)

[Multiple points – including and excluding revisions](#)

[The revision range for a single revision](#)

[Triple-dot notation](#)

[Searching history](#)

[Limiting the number of revisions](#)

[Matching revision metadata](#)

[Time-limiting options](#)

[Matching commit contents](#)

[Commit parents](#)

[Searching changes in revisions](#)

[Selecting types of change](#)

[History of a file](#)

[Path limiting](#)

[History simplification](#)

[Blame – the line-wise history of a file](#)

[Finding bugs with git bisect](#)

[Selecting and formatting the git log output](#)

[Predefined and user defined output formats](#)

[Including, formatting, and summing up changes](#)

[Summarizing contributions](#)

[Viewing a revision and a file at revision](#)

[Summary](#)

[3. Developing with Git](#)

[Creating a new commit](#)

[The DAG view of creating a new commit](#)

[The index – a staging area for commits](#)

[Examining the changes to be committed](#)

[The status of the working directory](#)

[Examining differences from the last revision](#)

[Unified Git diff format](#)

[Selective commit](#)

[Selecting files to commit](#)

[Interactively selecting changes](#)

[Creating a commit step by step](#)

[Amending a commit](#)

[Working with branches](#)

[Creating a new branch](#)

[Creating orphan branches](#)

[Selecting and switching to a branch](#)

[Obstacles to switching to a branch](#)

[Anonymous branches](#)

[Git checkout DWIM-mery](#)

[Listing branches](#)

[Rewinding or resetting a branch](#)

[Deleting a branch](#)

[Changing the branch name](#)

[Summary](#)

[4. Managing Your Worktree](#)

[Ignoring files](#)

[Marking files as intentionally untracked](#)

[Which types of file should be ignored?](#)

[Listing ignored files](#)

[Ignoring changes in tracked files](#)

[File attributes](#)

[Identifying binary files and end-of-line conversions](#)

[Diff and merge configuration](#)

[Generating diffs and binary files](#)

[Configuring diff output](#)

[Performing a 3-way merge](#)

[Transforming files \(content filtering\)](#)

[Obligatory file transformations](#)

[Keyword expansion and substitution](#)

[Other built-in attributes](#)

[Defining attribute macros](#)

[Fixing mistakes with the reset command](#)

[Rewinding the branch head, softly](#)

[Removing or amending a commit](#)

[Squashing commits with reset](#)

[Resetting the branch head and the index](#)

[Splitting a commit with reset](#)

[Saving and restoring state with the WIP commit](#)

[Discarding changes and rewinding branch](#)

[Moving commits to a feature branch](#)

[Undoing a merge or a pull](#)

[Safer reset – keeping your changes](#)

[Rebase changes to an earlier revision](#)

[Stashing away your changes](#)

[Using git stash](#)

[Stash and the staging area](#)

[Stash internals](#)

[Un-applying a stash](#)

[Recovering stashes that were dropped erroneously](#)

[Managing worktrees and the staging area](#)

[Examining files and directories](#)

[Searching file contents](#)

[Un-tracking, un-staging, and un-modifying files](#)

[Resetting a file to the old version](#)

[Cleaning the working area](#)

[Multiple working directories](#)

[Summary](#)

[5. Collaborative Development with Git](#)

[Collaborative workflows](#)

[Bare repositories](#)

[Interacting with other repositories](#)

[The centralized workflow](#)

[The peer-to-peer or forking workflow](#)

[The maintainer or integration manager workflow](#)

[The hierarchical or dictator and lieutenants workflows](#)

[Managing remote repositories](#)

[The origin remote](#)

[Listing and examining remotes](#)

[Adding a new remote](#)

[Updating information about remotes](#)

[Renaming remotes](#)

[Changing the remote URLs](#)

[Changing the list of branches tracked by remote](#)

[Setting the default branch of remote](#)

[Deleting remote-tracking branches](#)

[Support for triangular workflows](#)

[Transport protocols](#)

[Local transport](#)

[Smart transports](#)

[Native Git protocol](#)

[SSH protocol](#)

[Smart HTTP\(S\) protocol](#)

[Offline transport with bundles](#)

[Cloning and updating with bundle](#)

[Using bundle to update an existing repository](#)

[Utilizing bundle to help with the initial clone](#)

[Remote transport helpers](#)

[Transport relay with remote helpers](#)

[Using foreign SCM repositories as remotes](#)

Credentials/password management

Asking for passwords

Public key authentication for SSH

Credential helpers

Publishing your changes upstream

Pushing to a public repository

Generating a pull request

Exchanging patches

Chain of trust

Content-addressed storage

Lightweight, annotated, and signed tags

Lightweight tags

Annotated tags

Signed tags

Publishing tags

Tag verification

Signed commits

Merging signed tags (merge tags)

Summary

6. Advanced Branching Techniques

Types and purposes of branches

Long-running, perpetual branches

Integration, graduation, or progressive-stability branches

Per-release branches and per-release maintenance

Hotfix branches for security fixes

Per-customer or per-deployment branches

Automation branches

Mob branches for anonymous push access

The orphan branch trick

Short-lived branches

Topic or feature branches

Bugfix branches

Detached HEAD – the anonymous branch

Branching workflows and release engineering

The release and trunk branches workflow

The graduation, or progressive-stability branches workflow

The topic branches workflow

Graduation branches in a topic branch workflow

Branch management for a release in a topic branch workflow

Git-flow – a successful Git branching model

Fixing a security issue

Interacting with branches in remote repositories

Upstream and downstream

Remote-tracking branches and refspec

Remote-tracking branches

Refspec – remote to local branch mapping specification

Fetching and pulling versus pushing

Pull – fetch and update current branch

Pushing to the current branch in a nonbare remote repository

The default fetch refspec and push modes

Fetching and pushing branches and tags

Fetching branches

Fetching tags and automatic tag following

Pushing branches and tags

Push modes and their use

The simple push mode – the default

The matching push mode for maintainers

The upstream push mode for the centralized workflow

The current push mode for the blessed repository workflow

Summary

7. Merging Changes Together

Methods of combining changes

Merging branches

No divergence – fast-forward and up-to-date cases

Creating a merge commit

Merge strategies and their options

Reminder – merge drivers

Reminder – signing merges and merging tags

Copying and applying a changeset

Cherry-pick – creating a copy of a changeset

Revert – undoing an effect of a commit

Applying a series of commits from patches

[Cherry-picking and reverting a merge](#)

[Rebasing a branch](#)

[Merge versus rebase](#)

[Types of rebase](#)

[Advanced rebasing techniques](#)

[Resolving merge conflicts](#)

[The three-way merge](#)

[Examining failed merges](#)

[Conflict markers in the worktree](#)

[Three stages in the index](#)

[Examining differences – the combined diff format](#)

[How do we get there: git log --merge](#)

[Avoiding merge conflicts](#)

[Useful merge options](#)

[Rerere – reuse recorded resolutions](#)

[Dealing with merge conflicts](#)

[Aborting a merge](#)

[Selecting ours or theirs version](#)

[Scriptable fixes – manual file remerging](#)

[Using graphical merge tools](#)

[Marking files as resolved and finalizing merges](#)

[Resolving rebase conflicts](#)

[git-imerge – incremental merge and rebase for git](#)

[Summary](#)

[8. Keeping History Clean](#)

[An introduction to Git internals](#)

[Git objects](#)

[The plumbing and porcelain Git commands](#)

[Environment variables used by Git](#)

[Environment variables affecting global behavior](#)

[Environment variables affecting repository locations](#)

[Environment variables affecting committing](#)

[Rewriting history](#)

[Amending the last commit](#)

[An interactive rebase](#)

[Reordering, removing, and fixing commits](#)

Squashing commits
Splitting commits
Testing each rebased commit
External tools – patch management interfaces
Scripted rewrite with the git filter-branch
 Running the filter-branch without filters
 Available filter types for filter-branch and their use
 Examples of using the git filter-branch
External tools for large-scale history rewriting
 Removing files from the history with BFG Repo Cleaner
 Editing the repository history with reposurgeon
The perils of rewriting published history
 The consequences of upstream rewrite
 Recovering from an upstream history rewrite
Amending history without rewriting
 Reverting a commit
 Reverting a faulty merge
 Recovering from reverted merges
 Storing additional information with notes
 Adding notes to a commit
 How notes are stored
 Other categories and uses of notes
 Rewriting history and notes
 Publishing and retrieving notes
 Using the replacements mechanism
 The replacements mechanism
 Example – joining histories with git replace
 Historical note – grafts
 Publishing and retrieving replacements

Summary

9. Managing Subprojects – Building a Living Framework

Managing library and framework dependencies
Managing dependencies outside Git
Manually importing the code into your project
A Git subtree for embedding the subproject code
Creating a remote for a subproject

[Adding a subproject as a subtree](#)

[Cloning and updating superprojects with subtrees](#)

[Getting updates from subprojects with a subtree merge](#)

[Showing changes between a subtree and its upstream](#)

[Sending changes to the upstream of a subtree](#)

[The Git submodules solution: repository inside repository](#)

[Gitlinks, .git files, and the git submodule command](#)

[Adding a subproject as a submodule](#)

[Cloning superprojects with submodules](#)

[Updating submodules after superproject changes](#)

[Examining changes in a submodule](#)

[Getting updates from the upstream of the submodule](#)

[Sending submodule changes upstream](#)

[Transforming a subfolder into a subtree or submodule](#)

[Subtrees versus submodules](#)

[Use cases for subtrees](#)

[Use cases for submodules](#)

[Third-party subproject management solutions](#)

[Managing large Git repositories](#)

[Handling repositories with a very long history](#)

[Using shallow clones to get truncated history](#)

[Cloning only a single branch](#)

[Handling repositories with large binary files](#)

[Splitting the binary asset folder into a separate submodule](#)

[Storing large binary files outside the repository](#)

[Summary](#)

[10. Customizing and Extending Git](#)

[Git on the command line](#)

[Git-aware command prompt](#)

[Command-line completion for Git](#)

[Autocorrection for Git commands](#)

[Making the command line prettier](#)

[Alternative command line](#)

[Graphical interfaces](#)

[Types of graphical tools](#)

[Graphical diff and merge tools](#)

[Graphical interface examples](#)

[Configuring Git](#)

[Command-line options and environment variables](#)

[Git configuration files](#)

[The syntax of Git configuration files](#)

[Accessing the Git configuration](#)

[Basic client-side configuration](#)

[The rebase and merge setup, configuring pull](#)

[Preserving undo information – the expiry of objects](#)

[Formatting and whitespace](#)

[Server-side configuration](#)

[Per-file configuration with gitattributes](#)

[Automating Git with hooks](#)

[Installing a Git hook](#)

[A template for repositories](#)

[Client-side hooks](#)

[Commit process hooks](#)

[Hooks for applying patches from e-mails](#)

[Other client-side hooks](#)

[Server-side hooks](#)

[Extending Git](#)

[Command aliases for Git](#)

[Adding new Git commands](#)

[Credential helpers and remote helpers](#)

[Summary](#)

[11. Git Administration](#)

[Repository maintenance](#)

[Data recovery and troubleshooting](#)

[Recovering a lost commit](#)

[Troubleshooting Git](#)

[Git on the server](#)

[Server-side hooks](#)

[The pre-receive hook](#)

[Push-to-update hook for pushing to nonbare repositories](#)

[The update hook](#)

[The post-receive hook](#)

[The post-update hook \(legacy mechanism\)](#)

[Using hooks to implement the Git-enforced policy](#)

[Enforcing the policy with server-side hooks](#)

[Early notices about policy violations with client-side hooks](#)

[Signed pushes](#)

[Serving Git repositories](#)

[Local protocol](#)

[SSH protocol](#)

[Anonymous Git protocol](#)

[Smart HTTP\(S\) protocol](#)

[Dumb protocols](#)

[Remote helpers](#)

[Tools to manage Git repositories](#)

[Tips and tricks for hosting repositories](#)

[Reducing the size taken by repositories](#)

[Speeding up smart protocols with pack bitmaps](#)

[Solving the large nonresumable initial clone problem](#)

[Augmenting development workflows](#)

[Summary](#)

[12. Git Best Practices](#)

[Starting a project](#)

[Dividing work into repositories](#)

[Selecting the collaboration workflow](#)

[Choosing which files to keep under version control](#)

[Working on a project](#)

[Working on a topic branch](#)

[Deciding what to base your work on](#)

[Splitting changes into logically separate steps](#)

[Writing a good commit message](#)

[Preparing changes for submission](#)

[Integrating changes](#)

[Submitting and describing changes](#)

[The art of the change review](#)

[Responding to reviews and comments](#)

[Other recommendations](#)

[Don't panic, recovery is almost always possible](#)

Don't change the published history

Numbering and tagging releases

Automate what is possible

Summary

A. Bibliography

Index

Git: Mastering Version Control

Git: Mastering Version Control

Learn everything you need to take full control of your workflow with Git with this curated Learning Path – dive in and transform the way you work

A course in three modules



BIRMINGHAM - MUMBAI

Git: Mastering Version Control

Copyright © 2016 Packt Publishing

All rights reserved. No part of this course may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this course to ensure the accuracy of the information presented. However, the information contained in this course is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this course.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this course by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Published on: October 2016

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78712-320-5

www.packtpub.com

Credits

Authors

Ferdinando Santacroce

Aske Olsson

Rasmus Voss

Jakub Narębski

Reviewers

Fabrizio Donina

Giovanni Giorgi

Giovanni Toraldo

Kenneth Geissshirt

Shashikant Vaishnav

Markus Maiwald

Content Development Editor

Arun Nadar

Graphics

Abhinash Sahu

Production Coordinator

Melwyn Dsa

Preface

Git is the clear leader in the new paradigm of distributed version control systems. Originally developed by Linus Torvalds as a source control management (SCM) system for the Linux kernel to replace the proprietary SCM BitKeeper, Git has since conquered most of the open source world and is also used by lots of organizations for their private/proprietary projects.

If you are reading this course, you are probably a software developer and a professional. What makes a good professional? Sure, culture and experience are a part of the answer, but there's more: a good professional is one who can master different tools, choose the best tool for the job at hand, and has the necessary discipline to develop good working habits.

Version control is one of the base skills for developers and Git is one of the right tools for the job. However, Git can't be compared to a screwdriver, a simple tool with only a basic function; Git provides a complete toolbox that can help you manage your own code, within which there are also other sharp tools that should be handled with caution.

The aim of this course is to help you to start using Git and its commands in the safest way possible, to get things done without causing any injuries. Having said this, you will not get the most from Git commands if you do not acquire the right habits; just as is the case with other tools, in the end, it is the craftsman who makes all the difference. This course is meticulously designed to help you gain deeper insights into Git's architecture and its underlying concepts, behavior, and best practices.

What this learning path covers

[Module 1](#), *Git Essentials*, cover all the basic topics of Git, thereby letting you start using it even if you have little or no experience with

versioning systems. It starts off with the installation and do your first commit. You will learn how to manage and organize code, remotely work with repositories, and few basic Git commands that will come in handy in difficult situations. If you are familiar with other versioning tools this module will help you migrate to Git without much effort. Finally it ends with few best practices and hints that would develop good habits that every developer should posses.

[Module 2](#), *Git Version Control Cookbook*, is designed to give you practical recipes for everyday Git usage. The recipes can be used directly or as an inspiration for you. This module will cover the Git data model through practical recipes and in-depth explanations so you get a deeper understanding of the internal workings of Git. The recipes also give you precise step-by-step instructions to various common and uncommon Git operations. The book can help ease your daily work with Git by providing recipes for common issues, useful tips and tricks, and in-depth clarifications of why and how they work.

[Module 3](#), *Mastering Git*, starts with a quick implementation example of using Git for the collaborative development of a sample project to establish the foundation knowledge of Git's operational tasks and concepts. Furthermore, as you progress through this module, subsequent chapters provide detailed descriptions of the various areas of usage: from the source code archaeology, through managing your own work, to working with other developers. It also covers the advanced branching, merging and collaboration techniques. You will also learn to customize and extend Git, manage subprojects, and take up the administrative side of Git. Version control topics are accompanied by in-detail description of relevant parts of Git architecture and behavior. Finally the module ends with the Git best practices which will make you proficient in using Git.

What you need for this learning path

To follow the examples used in this book, and make use of them on Git, you only need a computer and a valid Git installation. Git is available for free on every platform (such as Linux, Windows, and Mac OS X).

Who this learning path is for

This learning path is for software developers who want to become proficient at using the Git version control system. A basic understanding of any version control system would be beneficial. If you have some experience working with command lines, using Linux admin, or just using Unix and want to know more about Git, then this book is ideal for you.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail <feedback@packtpub.com>, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this book from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

You can also download the code files by clicking on the **Code Files** button on the book's webpage at the Packt Publishing website. This page can be accessed by entering the book's name in the **Search** box. Please note that you need to be logged in to your Packt account.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at

<https://github.com/PacktPublishing/Mastering-Git-Skills>. We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to <https://www.packtpub.com/books/content/support> and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at <copyright@packtpub.com> with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at <questions@packtpub.com>, and we will do our best to address the problem.

Part I. Module 1

Git Essentials

Create, merge, and distribute code with Git, the most powerful and flexible versioning system available

Chapter 1. Getting Started with Git

Whether you are a professional or an amateur developer, you've likely heard about the concept of version control. You may know that adding a new feature, fixing broken ones, or stepping back to a previous condition is a daily routine.

This implies the use of a powerful tool that can help you take care of your work, allowing you to move around your project quickly and without friction.

There are many tools for this job on the market, both proprietary and open source. Usually, you will find **Version Control Systems (VCS)** and **Distributed Version Control Systems (DVCS)**. Some examples of centralized tools are **Concurrent Version System (CVS)**, **Subversion (SVN)**, **Team Foundation Server (TFS)** and **Perforce**, while in DVCS, you can find **Bazaar**, **Mercurial**, and **Git**. The main difference between the two families is the constraint—in the centralized system—to have a remote server where get and put your files; if the network is down, you are in trouble. In DVCS instead, you can have or not a remote server (even more than one), but you can work offline, too. All your modifications are locally recorded so that you can sync them some other time. Today, Git is the DVCS that has gained public favor more than others, growing quickly from a niche tool to mainstream.

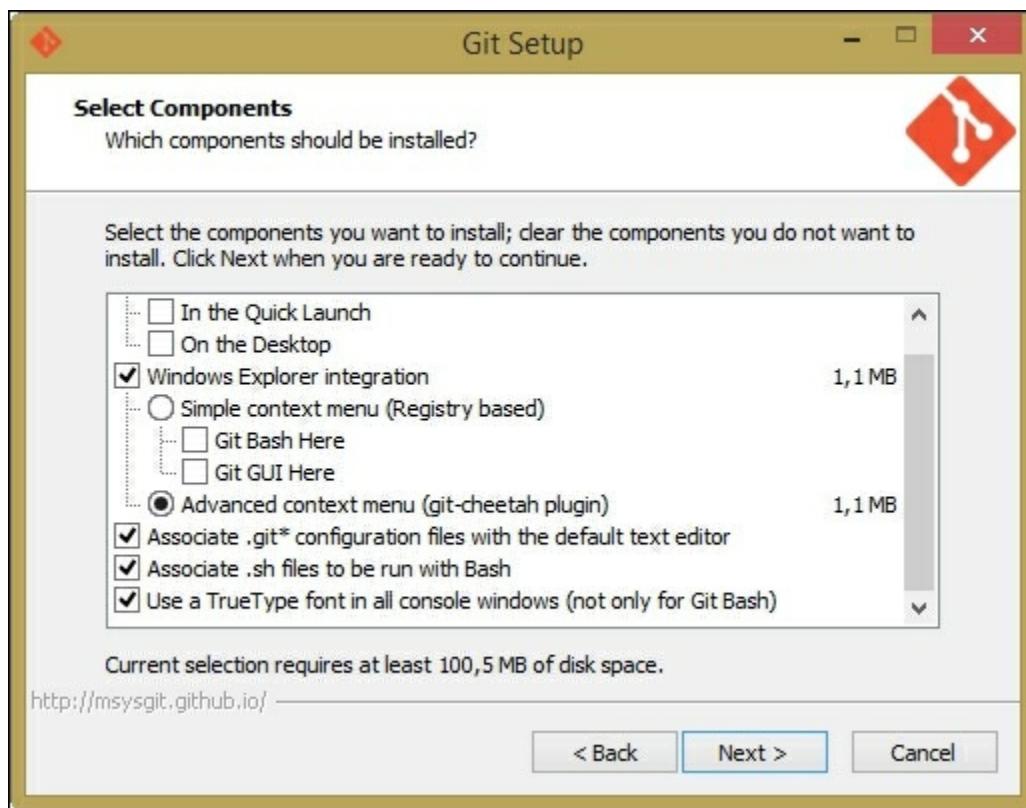
Git has rapidly grown as the de facto to version source code. It is the second famous child of *Linus Torvalds*, who, after creating the **Linux** kernel, forged this versioning software to keep trace of his millions lines of code.

In this first chapter, we will start at the very beginning, assuming that you do not have Git on your machine. This book is intended for developers who never used Git or used it a little bit, but who are scared to throw themselves headlong into it.

If you have never installed Git, this is your chapter. If you already have a working Git box, you can quickly read through it to check whether everything is right.

Installing Git

Git is open source software. If you are running a Windows machine, you can download it for free from <http://git-scm.com>. At the time of writing this book, the current version of Git is 1.9.5. If you are a Mac user, you can download it from <http://git-scm.com/downloads>; here, you can find the *.dmg file, too. Finally, if you are a Linux user, you probably have Git out of the box (if not, use the `apt-get install git` command or equivalent). Here, I won't go into too much detail about the installation process itself; I will only provide a few recommendations for Windows users shown in the following screenshot:

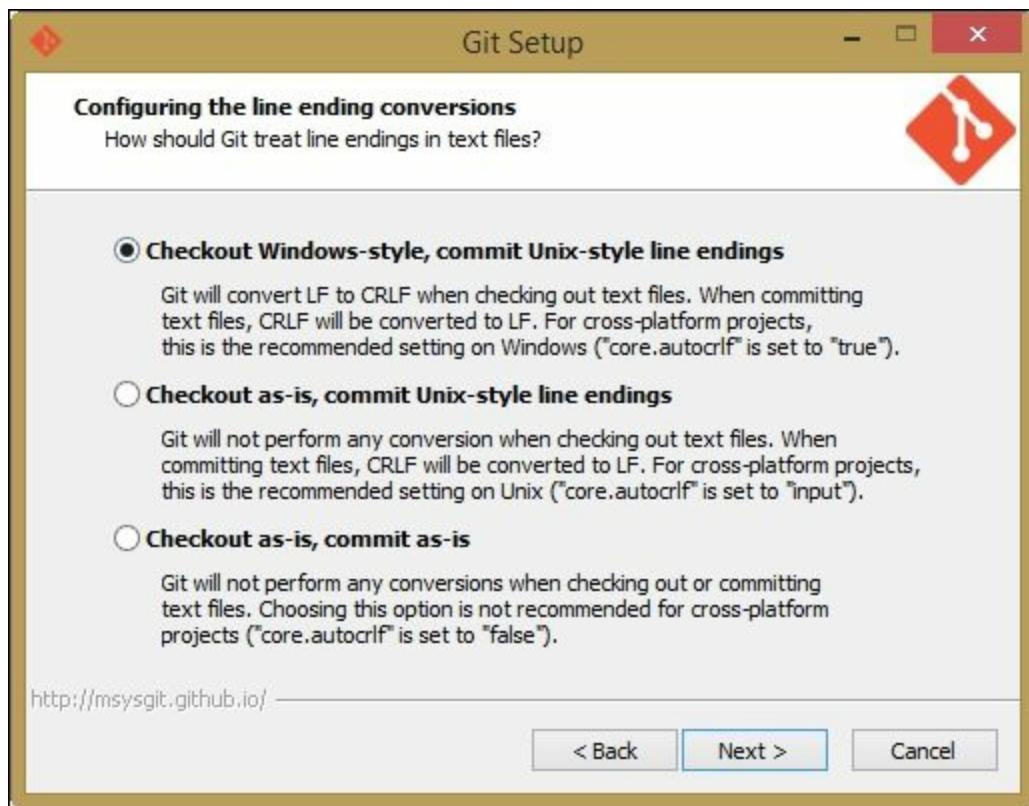


Enabling **Windows Explorer integration** is generally useful; you will benefit from a convenient way to open a Git prompt in any folder by right-clicking on the contextual menu.

Let Git be used in classic DOS command prompt, too, as shown in the following screenshot:



Git is provided with an embedded Windows-compatible version of the famous Bash shell from Linux, which we will use extensively. By doing this, we will also make Git available to third-party applications, such as GUIs and so on. It will come in handy when we give some GUI tools a try.



Use defaults for line endings. This will protect you from future annoyances.

At the end of the process, we will have Git installed, and all its *nix friends will be ready to use it.

Running our first Git command

Now, we have to test our installation. Is Git ready to rock? Let's find out!

Open a prompt and simply type `git` (or the equivalent, `git --help`), as shown in the following screenshot:

```
MINGW32:/c/Users/Nando
Welcome to Git (version 1.9.5-preview20141217)

Run 'git help git' to display the help index.
Run 'git help <command>' to display help for specific commands.

Nando@LIAN ~
$ git
usage: git [--version] [--help] [-C <path>] [-c name=value]
           [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
           [-p|--paginate|--no-pager] [--no-replace-objects] [--bare]
           [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
           <command> [<args>]

The most commonly used git commands are:
 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 Checkout a branch or paths to the working tree
 clone    Clone a repository into a new directory
 commit   Record changes to the repository
 diff     Show changes between commits, commit and working tree, etc
 fetch   Download objects and refs from another repository
 grep    Print lines matching a pattern
 init    Create an empty Git repository or reinitialize an existing one
 log     Show commit logs
 merge   Join two or more development histories together
 mv      Move or rename a file, a directory, or a symlink
 pull    Fetch from and 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
 reset   Reset current HEAD to the specified state
 rm      Remove files from the working tree and from the index
 show    Show various types of objects
 status  Show the working tree status
 tag     Create, list, delete or verify a tag object signed with GPG

'git help -a' and 'git help -g' lists available subcommands and some
concept guides. See 'git help <command>' or 'git help <concept>'
to read about a specific subcommand or concept.

Nando@LIAN ~
```

If Git has been installed correctly, typing `git` without specifying anything else will result in a short help page, with a list of common commands. If not, try reinstalling Git, ensuring that you have checked the **Use Git from the Windows Command Prompt** option. Otherwise,

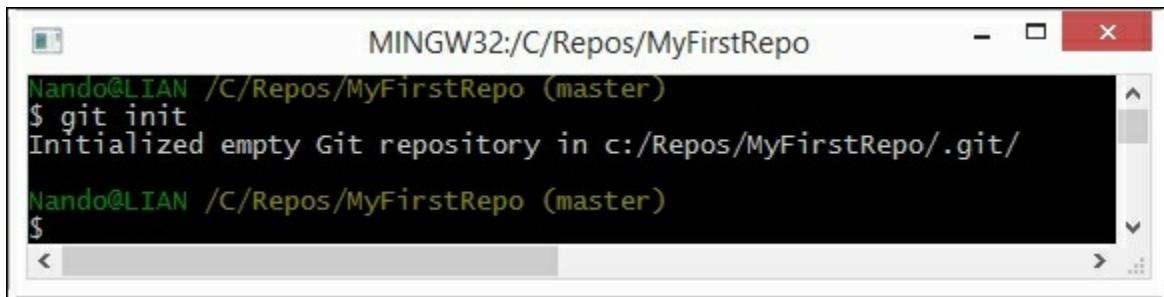
Git will be available only within the embedded Bash shell.

So, we have Git up and running! Are you excited? Let's begin to type!

Setting up a new repository

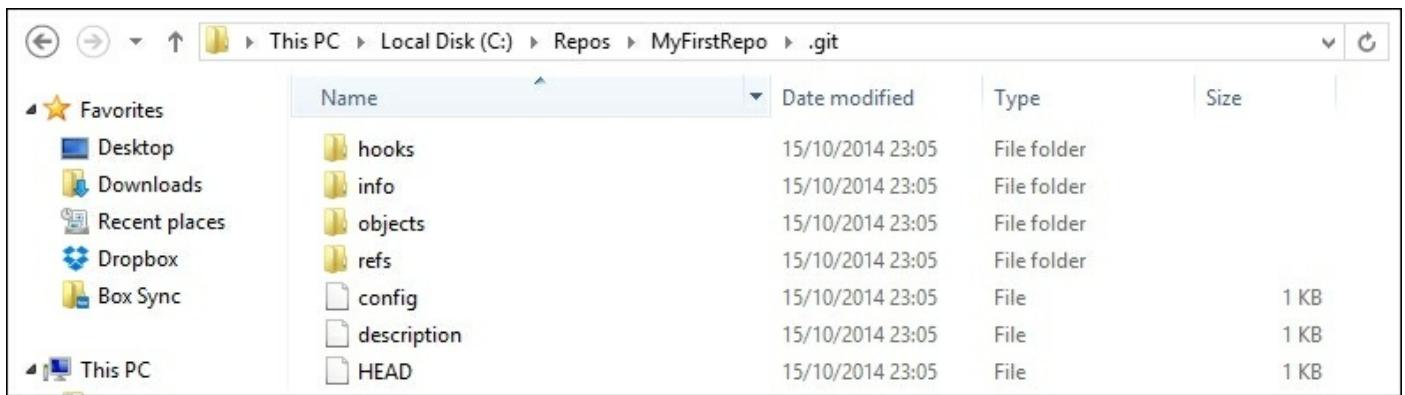
The first step is to set up a new repository (or repo, for short). A **repo** is a container for your entire project; every file or subfolder within it belongs to that repository, in a consistent manner. Physically, a repository is nothing other than a folder that contains a special `.git` folder, the folder where the magic happens.

Let's try to make our first repo. Choose a folder you like, and type the `git init` command, as shown here:



```
MINGW32:/C/Repos/MyFirstRepo
Nando@LIAN /C/Repos/MyFirstRepo (master)
$ git init
Initialized empty Git repository in c:/Repos/MyFirstRepo/.git/
Nando@LIAN /C/Repos/MyFirstRepo (master)
$
```

Whoa! What just happened? Git created a `.git` subfolder. The subfolder (normally hidden in Windows) contains some files and folders, as shown in the next screenshot:

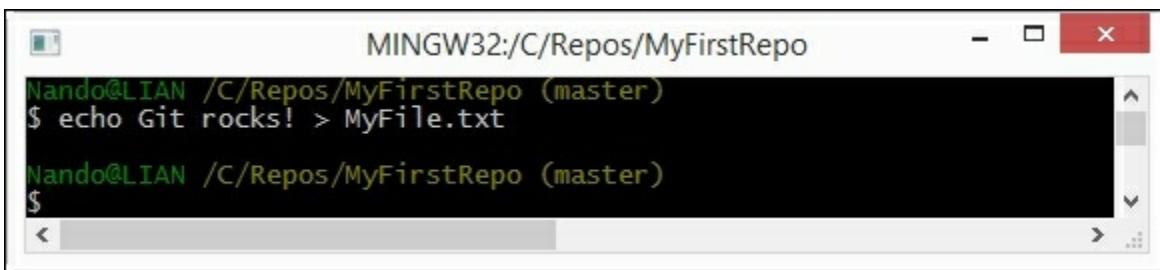


At this point, it is not important for us to understand what is inside this folder. The only thing you have to know is that you do not have to touch it, ever! If you delete it or if you modify files inside by hand, you could get into trouble. Have I frightened you enough?

Now that we have a repo, we can start to put files inside it. Git can trace the history of any gender of files, text based or binary, small or large, with the same efficiency (more or less, large files are always a problem).

Adding a file

Let's create a text file just to give it a try.

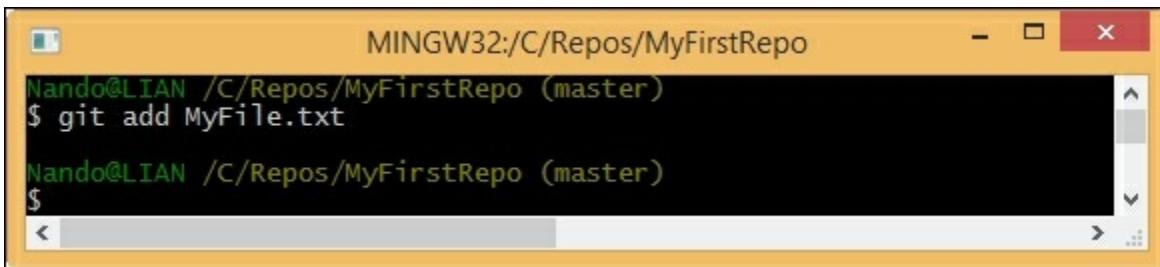


A screenshot of a terminal window titled "MINGW32:/C/Repos/MyFirstRepo". The window shows a command-line interface where the user has run the command "\$ echo Git rocks! > MyFile.txt". The output of this command is displayed below the command, indicating that the file was successfully created.

```
MINGW32:/C/Repos/MyFirstRepo
Nando@LIAN /C/Repos/MyFirstRepo (master)
$ echo Git rocks! > MyFile.txt
Nando@LIAN /C/Repos/MyFirstRepo (master)
$
```

And now what? Is that all? No! We have to tell Git to put this file in your repo, *explicitly*. Git doesn't do anything that you don't want it to. If you have some spare files or temp ones in your repo, Git will not be compatible with them, but will only remind you that there are some files in your repo that are not under version control (in the next chapter, we will see how to instruct Git to ignore them when necessary).

Ok, back to the topic. I want `MyFile.txt` under the control of Git, so let's add it, as shown here:



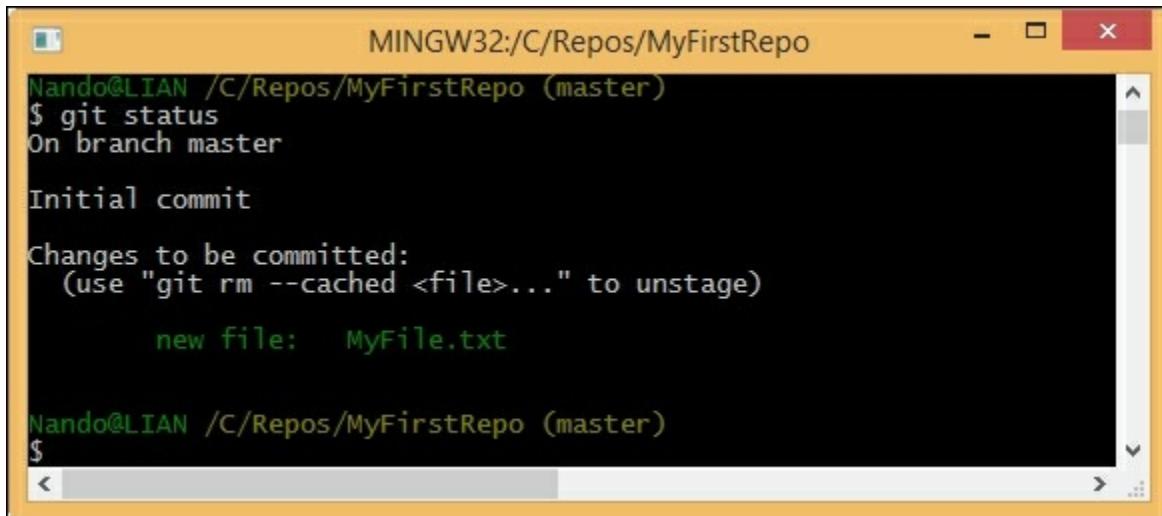
A screenshot of a terminal window titled "MINGW32:/C/Repos/MyFirstRepo". The window shows a command-line interface where the user has run the command "\$ git add MyFile.txt". The output of this command is displayed below the command, indicating that the file was added to the repository.

```
MINGW32:/C/Repos/MyFirstRepo
Nando@LIAN /C/Repos/MyFirstRepo (master)
$ git add MyFile.txt
Nando@LIAN /C/Repos/MyFirstRepo (master)
$
```

The `git add` command tells Git that we want it to take care of that file and check it for future modifications.

Has Git obeyed us? Let's see.

Using the `git status` command, we can check the status of the repo, as shown in the following screenshot:



A screenshot of a terminal window titled "MINGW32:/C/Repos/MyFirstRepo". The window shows the following text output:

```
Nando@LIAN /C/Repos/MyFirstRepo (master)
$ git status
On branch master
Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

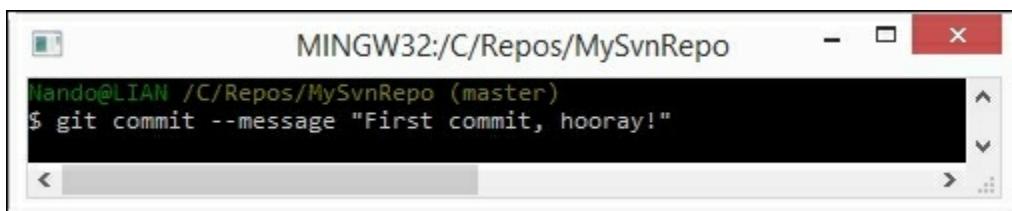
    new file:  MyFile.txt

Nando@LIAN /C/Repos/MyFirstRepo (master)
$
```

As we can see, Git has accomplished its work as expected. In this image, we can read words such as `branch`, `master`, `commit` and `unstage`. We will look at them briefly, but for the moment, let's ignore them.

Commit the added file

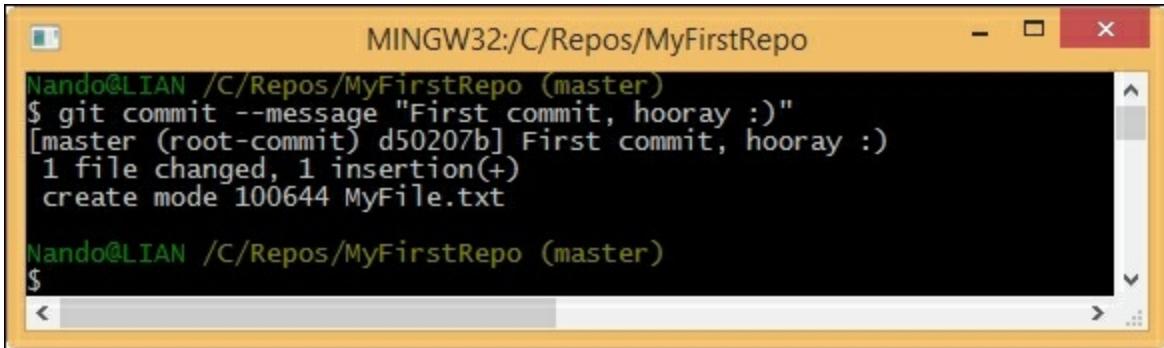
At this point, Git knows about `MyFile.txt`, but we have to perform another step to fix the snapshot of its content. We have to commit it using the appropriate `git commit` command. This time, we will add some flavor to our command, using the `--message` (or `-m`) subcommand, as shown here:



A screenshot of a terminal window titled "MINGW32:/C/Repos/MySvnRepo". The window shows the following text output:

```
Nando@LIAN /C/Repos/MySvnRepo (master)
$ git commit --message "First commit, hooray!"
```

Press the *Enter* key.



The screenshot shows a terminal window titled "MINGW32:/C/Repos/MyFirstRepo". The command \$ git commit --message "First commit, hooray :)" was run, resulting in a commit message "[master (root-commit) d50207b] First commit, hooray :)". The output also indicates "1 file changed, 1 insertion(+)" and "create mode 100644 MyFile.txt". The terminal prompt Nando@LIAN /C/Repos/MyFirstRepo (master)\$ is visible at the bottom.

Feel the magic—a new branch is born!

With the commit of `MyFile.txt`, we have finally fired up our repo. It now has a `master` branch with a file within it. We will play with branches in the forthcoming chapters. Right now, think of it as a course of our repository, and keep in mind that a repository can have multiple courses that often cross each other.

Modify a committed file

Now, we can try to make some modifications to the file and see how to deal with it in the following screenshot:

```
Nando@LIAN /C/Repos/MyFirstRepo (master)
$ git commit --message "First commit, hooray :)"
[master (root-commit) d50207b] First commit, hooray :)
 1 file changed, 1 insertion(+)
 create mode 100644 MyFile.txt

Nando@LIAN /C/Repos/MyFirstRepo (master)
$ echo "I didn't think it was that easy :0" >> MyFile.txt

Nando@LIAN /C/Repos/MyFirstRepo (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   MyFile.txt

no changes added to commit (use "git add" and/or "git commit -a")

Nando@LIAN /C/Repos/MyFirstRepo (master)
$
```

As you can see, Bash shell warns us that there are some modifications painting the name of the modified files in red. Here, the `git status` command informs us that there is a file with some modifications and that we need to commit it if we want to save this modification step in the repository history.

However, what does `no changes added to commit` mean? It is simple. Git makes you take a second look at what you want to include in the next commit. If you have touched two files but you want to commit only one, you can add only that one.

If you try to do a commit without skipping the `add` step, nothing will happen. We will see this behavior in depth in the next chapter.

So, let's add the file again for the purpose of getting things ready for the next commit.

```
MINGW32:/C/Repos/MyFirstRepo
no changes added to commit (use "git add" and/or "git commit -a")
Nando@LIAN /C/Repos/MyFirstRepo (master)
$ git commit
On branch master
Changes not staged for commit:
  modified:  MyFile.txt

no changes added to commit

Nando@LIAN /C/Repos/MyFirstRepo (master)
$ git add .

Nando@LIAN /C/Repos/MyFirstRepo (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:  MyFile.txt

Nando@LIAN /C/Repos/MyFirstRepo (master)
$
```

Let's do another commit, this time, avoiding the `--message` subcommand. So, type `git commit` and hit the *Enter* key.

```
MINGW32:/C/Repos/MyFirstRepo
Nando@LIAN /C/Repos/MyFirstRepo (master)
$ git commit
```

Fasten your seatbelts! You are now entering into a piece of code history!

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#     modified:   MyFile.txt
#
#
#
c:\Repos\MyFirstRepo\.git\COMMIT_EDITMSG [unix] (07:53 26/02/2015)1,0-1 ALL
```

What is that? It's **Vim (Vi Improved)**, an ancient and powerful text editor. You can configure Git to use your own preferred editor, but if you don't do it, this is what you have to deal with. Vim is powerful, but for newcomers, it can be a pain to use. It has a strange way of dealing with text, so to start typing, you have to press `i` for inserting text, as shown in the following screenshot:

```
This thing is becoming serious :D
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#     modified:   MyFile.txt
#
#
#
<Repos\MyFirstRepo\.git\COMMIT_EDITMSG[+] [unix] (07:53 26/02/2015)1,34 ALL
-- INSERT --
```

Once you have typed your commit message, you can press `Esc` to get out of the editing mode. Then, you can type the `:w` command to write changes and the `:q` command to quit. You can also type the command in pairs as `:wq`.

```
COMMIT_EDITMSG + (c:\Repos\MyFirstRepo\.git) - VIM
This thing is becoming serious :D
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#       modified:  MyFile.txt
#
<Repos\MyFirstRepo\.git\COMMIT_EDITMSG[+] [unix] (07:53 26/02/2015)1,33 ATI
:wq
```

After that, press *Enter* and another commit is done, as shown here:

```
Nando@LIAN /C/Repos/MyFirstRepo (master)
$ git commit
[master 9efda35] This thing is becoming serious :D
 1 file changed, 1 insertion(+)

Nando@LIAN /C/Repos/MyFirstRepo (master)
$
```

Note that when you saved the commit message in Vim, Git automatically dispatches the commit work, as you can see in the preceding screenshot.

Well done! Now, it's time to recap.

Summary

In this chapter, you learned that Git is not so difficult to install, even on a non-Unix platform such as Windows.

Once you have chosen a directory to include in a Git repository, you can see that initializing a new Git repository is as easy as executing a `git init` command, and nothing more. Don't worry now about saving it on a remote server and so on. It's not mandatory to save it; you can do this when you need to, preserving the entire history of your repo. This is a killer feature of Git and DVCS in general. You can comfortably work offline and push your work to a remote location when the network is available, without hassle.

In the end, we discovered one of the most important character traits of Git: it will do nothing if you don't mention it explicitly. You also learned a little bit about the `add` command. We were obliged to perform a `git add` command for a file when we committed it to Git the very first time. Then, we used another command when we modified it. This is because if you modify a file, Git does not expect that you want it to be automatically added to the next commit (and it's right, I'll say).

In the next chapter, we will discover some fundamentals of Git.

Chapter 2. Git Fundamentals – Working Locally

In this chapter, we will go deep into some of the fundamentals of Git. It is essential to understand how Git thinks about files, its way of tracking the history of commits, and all the basic commands that we need to master in order to become proficient.

Repository structure and file status life cycle

The first thing to understand while working with Git is how it manages files and folders within the repository. This is the time to analyze a default repository structure.

The working directory

In [Chapter 1, Getting Started with Git](#), we created an empty folder and initialized a new repository using the `git init` command (in `C:\Repos\MyFirstRepo`). Starting from now, we will call this folder the **working directory**. A folder that contains an initialized Git repository is a working directory. You can move the working directory around your file system without losing or corrupting your repository.

Within the working directory, you also learned that there is a `.git` directory. Let's call it the **git directory** from now on. In the git directory there are files and folders that compose our repository. Thanks to this, we can track the file status, configure the repository, and so on.

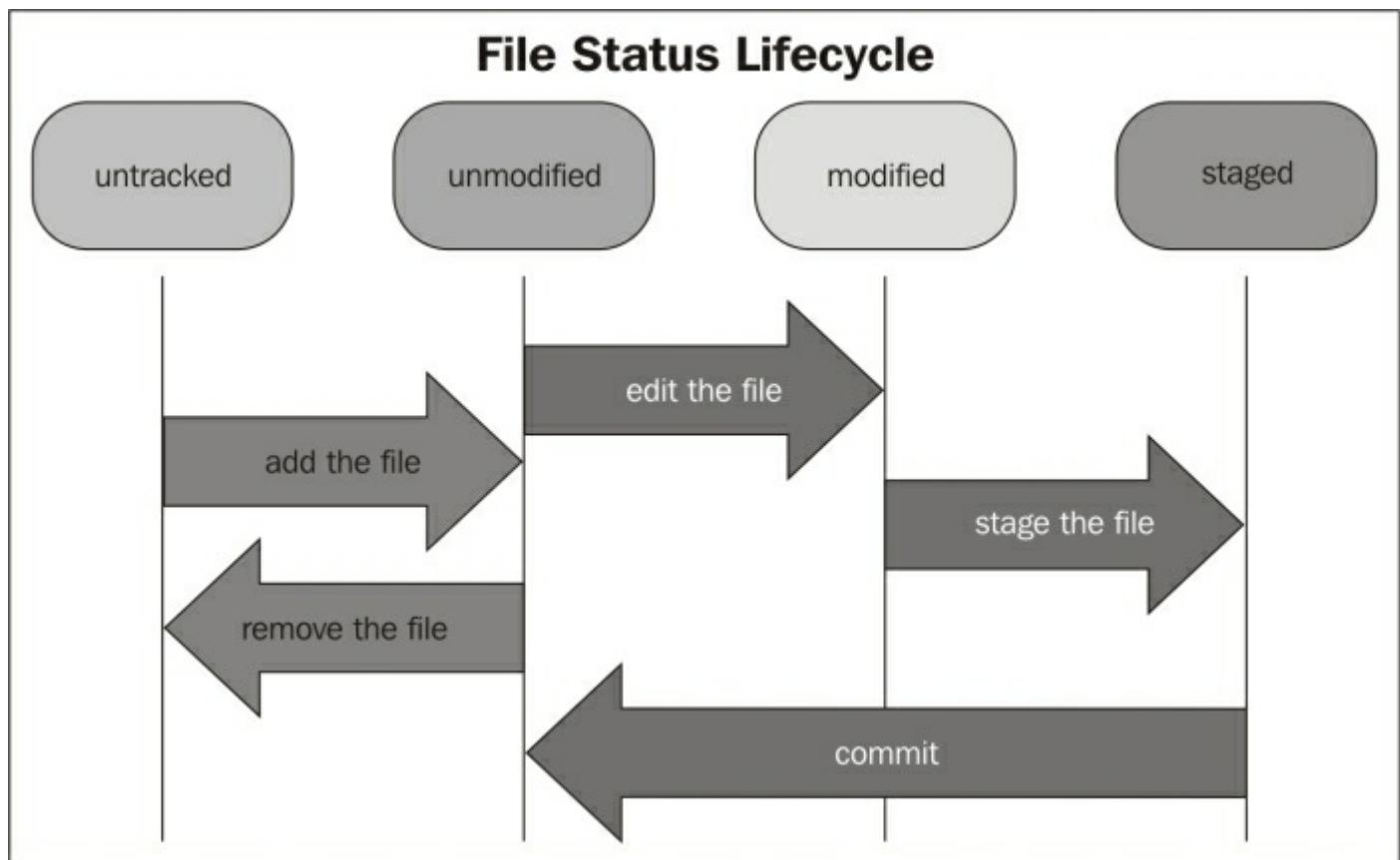
File statuses

In [Chapter 1, Getting Started with Git](#), we used two different commands: `git add` and `git commit`. These commands allowed us to change the status of a file, making Git change its status from "I don't

know who you are" to "You are in a safe place".

When you create or copy a new file in the working directory, the first state of the file is **untracked**. This means that Git sees that there is something new, but it won't take care of it (it would not track the new file). If you want to include the file in your repository, you have to add it using the `add` command. Once it is added, the state of the file becomes **unmodified**. It means that the file is new (Git says it is unmodified because it never tracked changes earlier) and ready to be committed, or it has reached the **staging area** (also called **index**). If you modify a file that is already added to the index, it changes its status to **modified**.

The following screenshot explains the file status life cycle:



The staging area

The staging area or index is a virtual place that collects all the files you want to include in the next commit. You will often hear people talk about staged files with regard to Git, so take care of this concept. All the

files (new or modified) you want to include in the next commit have to be staged using the `git add` command. If you staged a file accidentally, you have to unstage it to remove it from the next commit bundle. Unstaging is not difficult; you can do it in many ways. Let me explain a few concepts. This *several ways to do the same thing* is an organic problem of Git. Its constant and fast evolution sometimes increases confusion, resulting in different commands that do the same thing. This is because it will not penalize people used to working in a particular manner, allowing them the time for some Git revision to understand the new or better way. Fortunately, Git often suggests the best way to do what you want to do and warns you when you use obsolete commands. When in doubt, remember that there are **man pages**. You can obtain some useful suggestions by typing `git <command> --help` (-h for short) and seeing what the command is for and how to use it.

Unstaging a file

Well, back to our main topic. Before continuing, let's try to understand the unstaging concept better. Open the repo folder (C:\Repos\MyFirstRepo) in Bash and follow these simple steps:

1. Be sure to be in a clean state by typing `git status`. If Git says "nothing to commit, working directory clean," we are ready to start.
2. Create a new file `touch NewFile.txt`.
3. Using `git status` again, verify that `NewFile.txt` is untracked.
4. Add it to the index so that Git can start tracking it. So, use the `git add NewFile.txt` command and go on.
5. When done, use the suggested `git reset HEAD <file>` command to back the file in the untracked status.

```
MINGW32:/C/Repos/MyFirstRepo
Nando@LIAN /C/Repos/MyFirstRepo (master)
$ git status
On branch master
nothing to commit, working directory clean

Nando@LIAN /C/Repos/MyFirstRepo (master)
$ echo "This is a new file" >> NewFile.txt

Nando@LIAN /C/Repos/MyFirstRepo (master)
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    NewFile.txt

nothing added to commit but untracked files present (use "git add" to track)

Nando@LIAN /C/Repos/MyFirstRepo (master)
$ git add NewFile.txt
warning: LF will be replaced by CRLF in NewFile.txt.
The file will have its original line endings in your working directory.

Nando@LIAN /C/Repos/MyFirstRepo (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   NewFile.txt

Nando@LIAN /C/Repos/MyFirstRepo (master)
$ git reset HEAD NewFile.txt

Nando@LIAN /C/Repos/MyFirstRepo (master)
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    NewFile.txt

nothing added to commit but untracked files present (use "git add" to track)

Nando@LIAN /C/Repos/MyFirstRepo (master)
$
```

It should be clear now. It worked as expected: our file returned in the untracked state, as it was before the `add` command.

Tip

`Git reset` is a powerful command, and it can completely destroy your

actual work if used improperly. Do not play with it if you don't know exactly what you are doing.

Another way to unstage a file that you just added is to use the `git rm` command. If you want to preserve the file on your folder, you can use the `--cached` option. This option simply removes it from the index, but not from the filesystem. However, remember that `git rm` is to remove files from the index. So, if you use the `git rm` command on an already committed file, you actually mark it for deletion. The next commit will delete it.

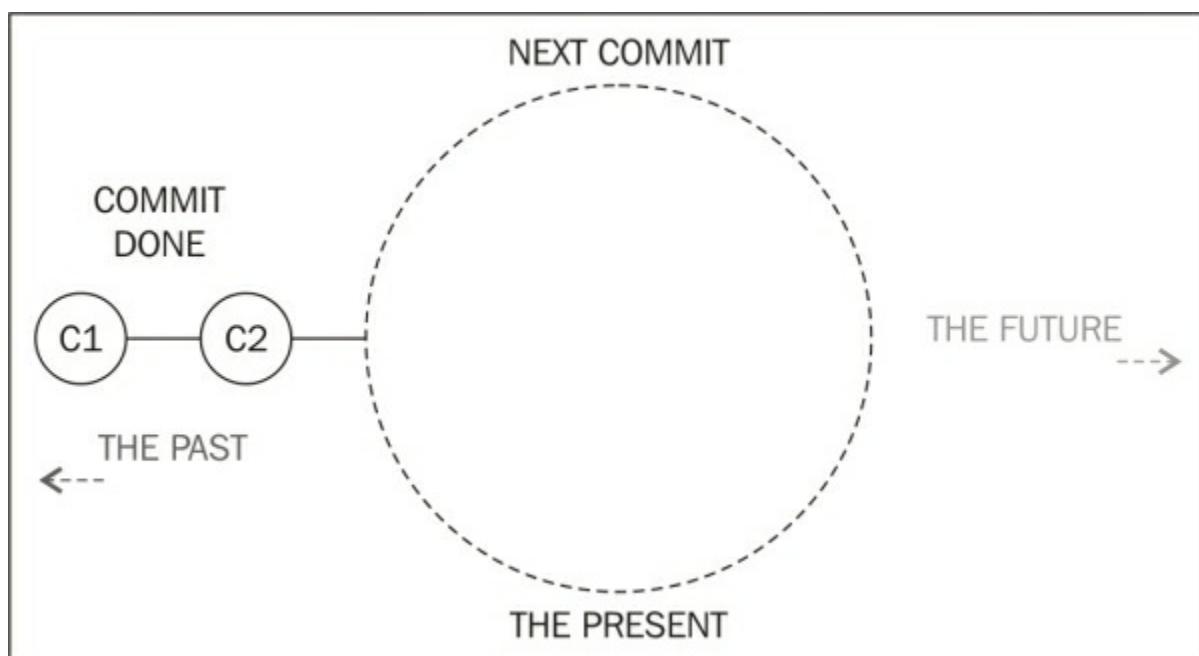
The time metaphor

Using the `git reset` command, we also get in touch with another fundamental of Git, the `HEAD` pointer. Let's try and understand this concept.

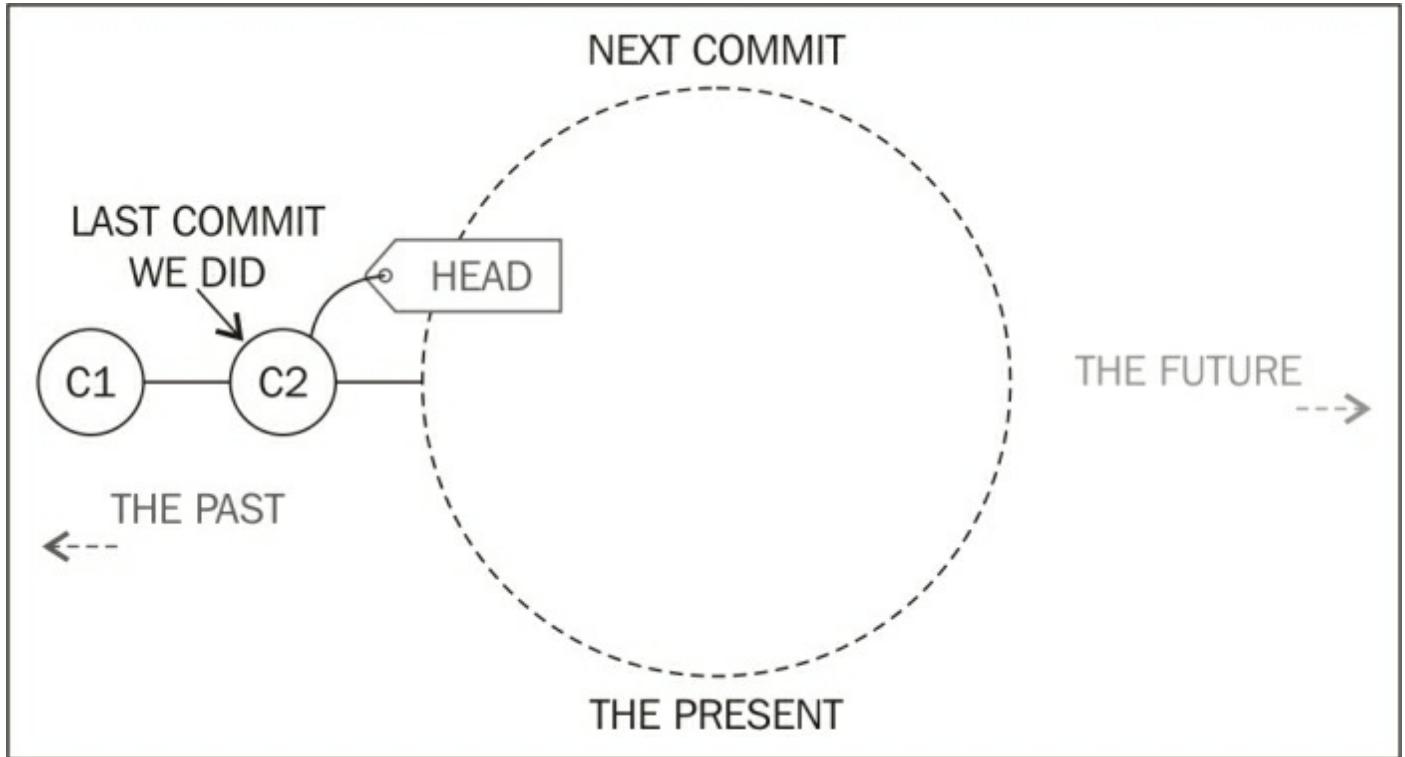
A repository is made of commits, as a life is made of days. Every time you commit something, you write a piece of the history.

The past

The past is represented by the previous commits that we did, as shown by **C1** and **C2** in the following diagram:



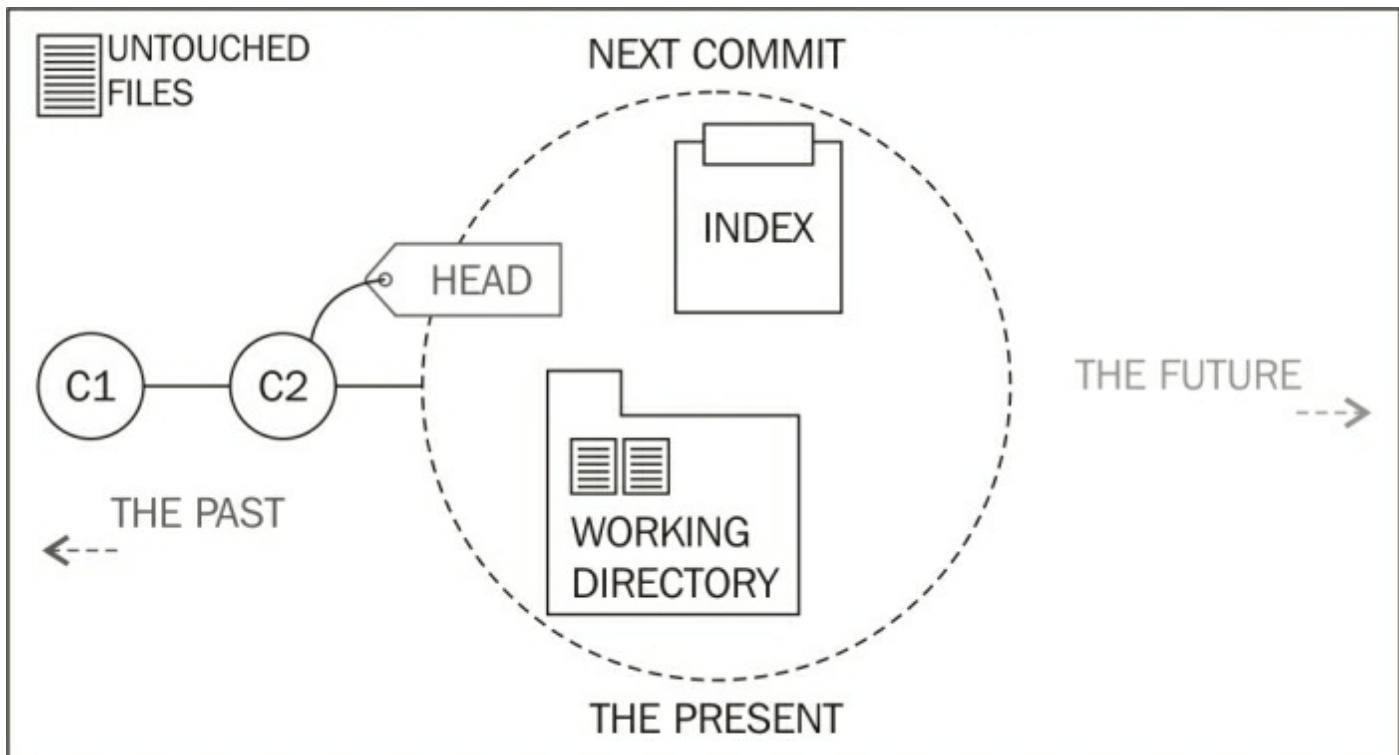
The `HEAD` pointer is the reference to the last commit we did or the parent of the next commit we will do, as shown in the diagram:



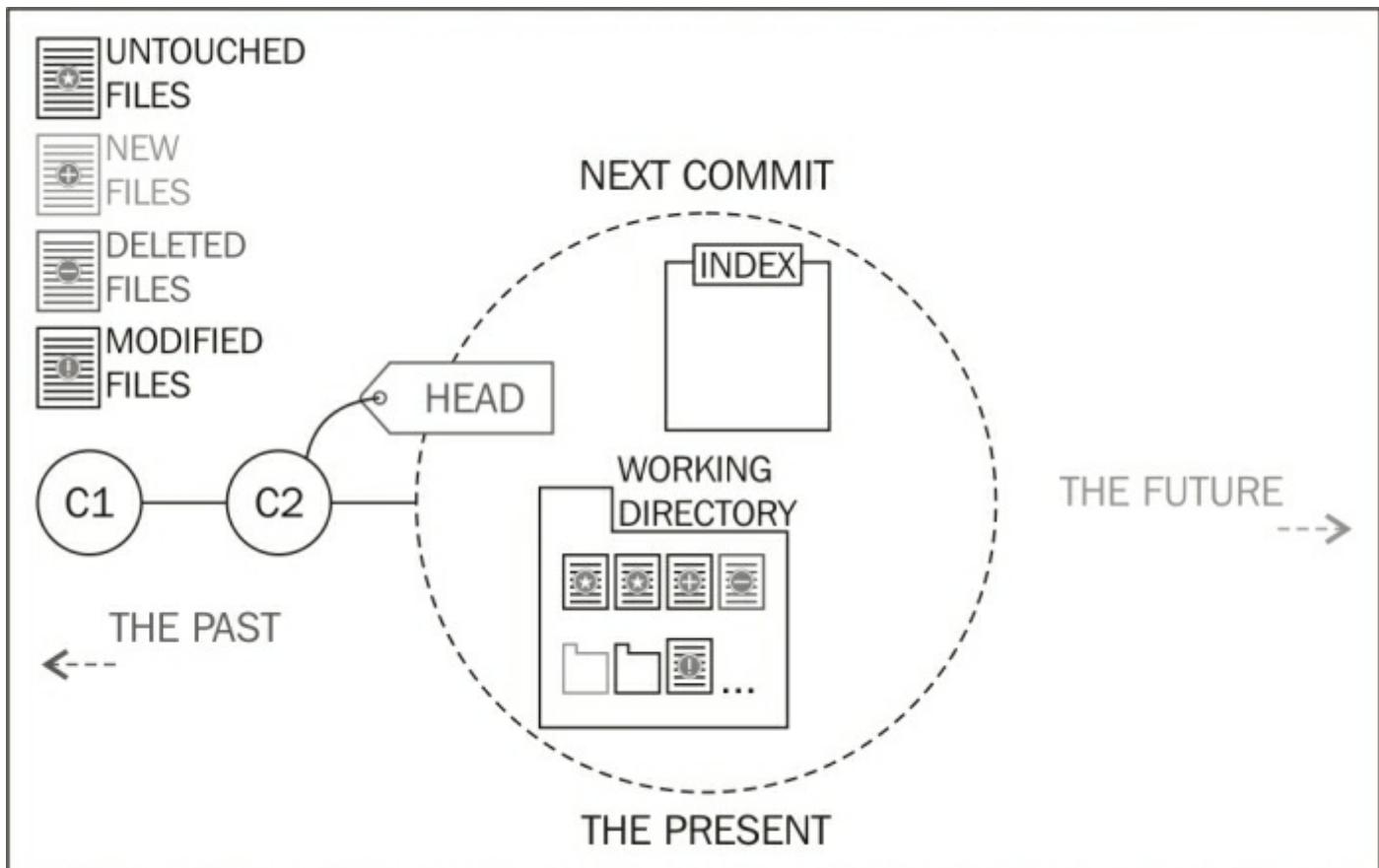
So, the `HEAD` pointer is the road sign that indicates the way to move one step back to the past.

The present

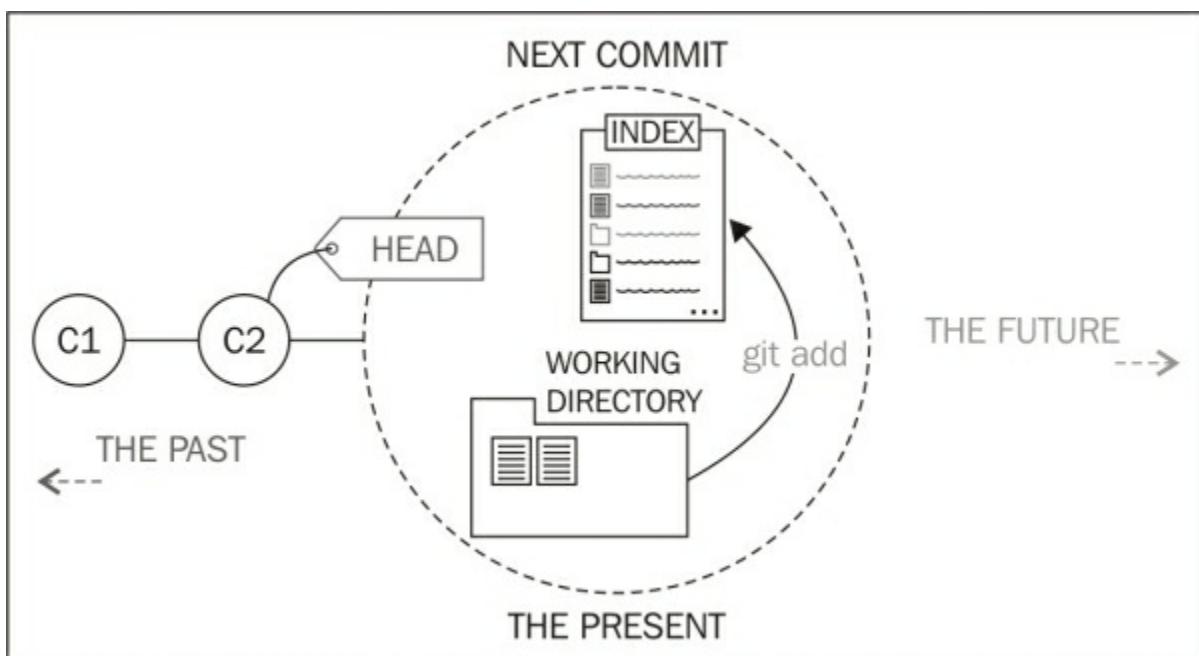
The present is where we work. When a previous commit is done, it becomes part of the past, and the present shows itself like this diagram:



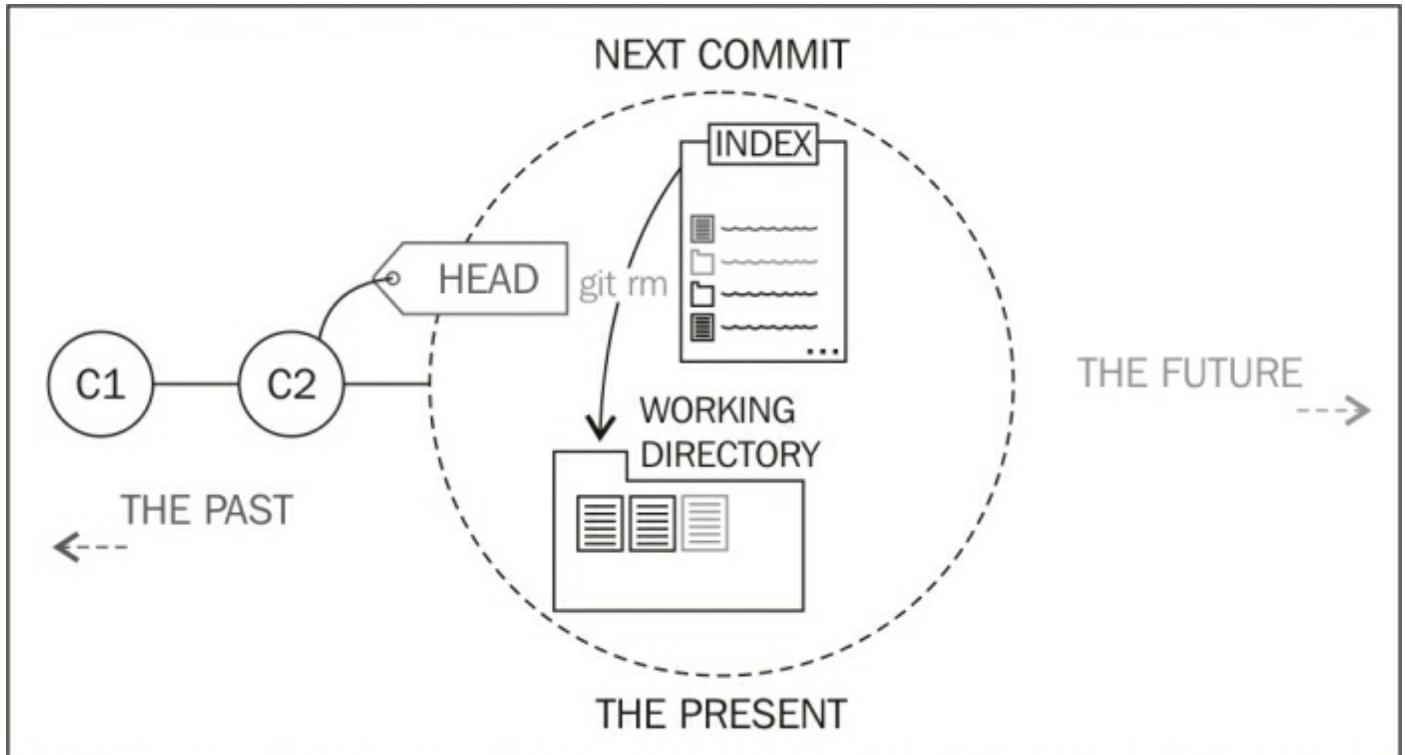
We have a `HEAD` reference that points out where we came from (the **C2** commit). Resetting to `HEAD` as we did earlier is a manner of going back in this initial state, where there are no modifications yet. Then, we have the working directory. This directory collects files added to the repository in the previous commits. Now, it is in the untouched state. Within this place, we do our work in files and folders, adding, removing, or modifying them.



Our work remains in the working directory until we decide to put it in the next commit we will perform. Using the `git add` command, we add what we want to promote to the next commit, marking them into the index, as shown in this diagram:

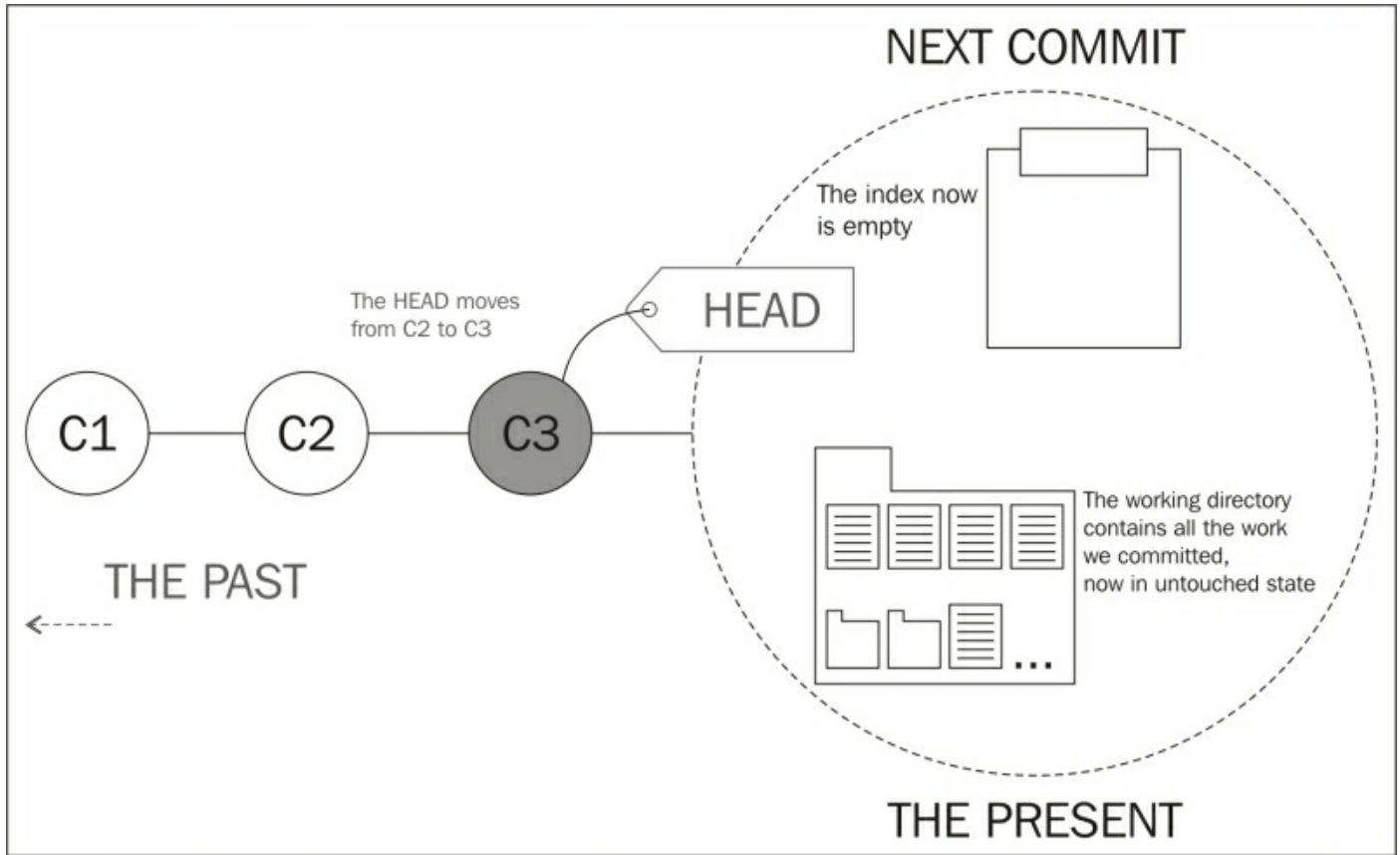


With `git rm --cached <file or folder>`, you can unstage a file by removing it from the index, as shown here:



With `git reset --hard HEAD`, we will go back to the initial state, losing all the changes we made in the working directory.

At the end, once you commit, the present becomes part of the past. The working directory comes back to the initial state, where all is untouched and the index is emptied, as shown in this diagram:



The future

What about the future? Well, even though Git is a very powerful tool, it can't predict the future, not for now at least.

Working with repositories

Let's get our hands dirty!

If you are reading this book, you are probably a programmer, as I am. Using a programming language and its source and binary files to exercise could be fine. However, I don't want to distract you from understanding a language you probably don't use every day. So, let's settle things once and for all. When needed, I will use a nice markup language called **Markdown** (see <http://daringfireball.net/projects/markdown/>).

Apart from being simple yet powerful, Markdown silently became the favorite choice while typing `readme` files (for example, GitHub uses it extensively) or comments in forums or other online discussion places, such as StackOverflow. Mastering it is not our goal, but to be able to do the basic things is surely a skill that can be useful in the future.

Before you proceed, create a new folder for exercises, for example, `C:\Repos\Exercises`. We will use different folders for different exercises.

Unstaging a file

Consider the following scenario. You are in a new repository located in `C:\Repos\Exercises\Ch1-1`. The working directory actually contains two files: `first.txt` and `second.txt`. You have accidentally put both `first.txt` and `second.txt` in the staging area, while you actually want to only commit `first.txt`. So, now:

- Remove the `second.txt` file from the index
- Commit the changes

The result of this is that only `first.txt` will be a part of the commit.

Follow these simple steps to solve this simple task:

1. Create the `C:\Repos\Exercises\Ch1-1` folder and open Bash inside it.

2. Use the `git init` command on the repository as we learned.
3. Create the `first.txt` and `second.txt` files and add them to the staging area, as shown in the following screenshot:

The screenshot shows a terminal window titled "MINGW32:/C/Repos/Exercises/Ch1-1". The session starts with the Git welcome message. Nando runs `git init`, which initializes an empty repository in the current directory. Next, he creates two files, `first.txt` and `second.txt`. He then runs `git status`, which shows he is on the `master` branch and has an initial commit. The output indicates there are untracked files: `first.txt` and `second.txt`. He uses `git add` to stage both files. After adding, he runs `git status` again, which now shows changes to be committed: `new file: first.txt` and `new file: second.txt`. Finally, he is at the prompt, ready for the next command.

```
MINGW32:/C/Repos/Exercises/Ch1-1
Welcome to Git (version 1.9.4-preview20140929)

Run 'git help git' to display the help index.
Run 'git help <command>' to display help for specific commands.

Nando@LIAN /C/Repos/Exercises/Ch1-1
$ git init
Initialized empty Git repository in c:/Repos/Exercises/Ch1-1/.git/
Nando@LIAN /C/Repos/Exercises/Ch1-1 (master)
$ touch first.txt second.txt

Nando@LIAN /C/Repos/Exercises/Ch1-1 (master)
$ git status
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    first.txt
    second.txt

nothing added to commit but untracked files present (use "git add" to track)

Nando@LIAN /C/Repos/Exercises/Ch1-1 (master)
$ git add first.txt second.txt

Nando@LIAN /C/Repos/Exercises/Ch1-1 (master)
$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

    new file:  first.txt
    new file:  second.txt

Nando@LIAN /C/Repos/Exercises/Ch1-1 (master)
$
```

At this point, remove the `second.txt` file and commit. This time, try to use the handy `git rm --cached` command, as suggested here:

```
MINGW32:/C/Repos/Exercises/Ch1-1
Nando@LIAN /C/Repos/Exercises/Ch1-1 (master)
$ git rm --cached second.txt
rm 'second.txt'

Nando@LIAN /C/Repos/Exercises/Ch1-1 (master)
$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

    new file:  first.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    second.txt

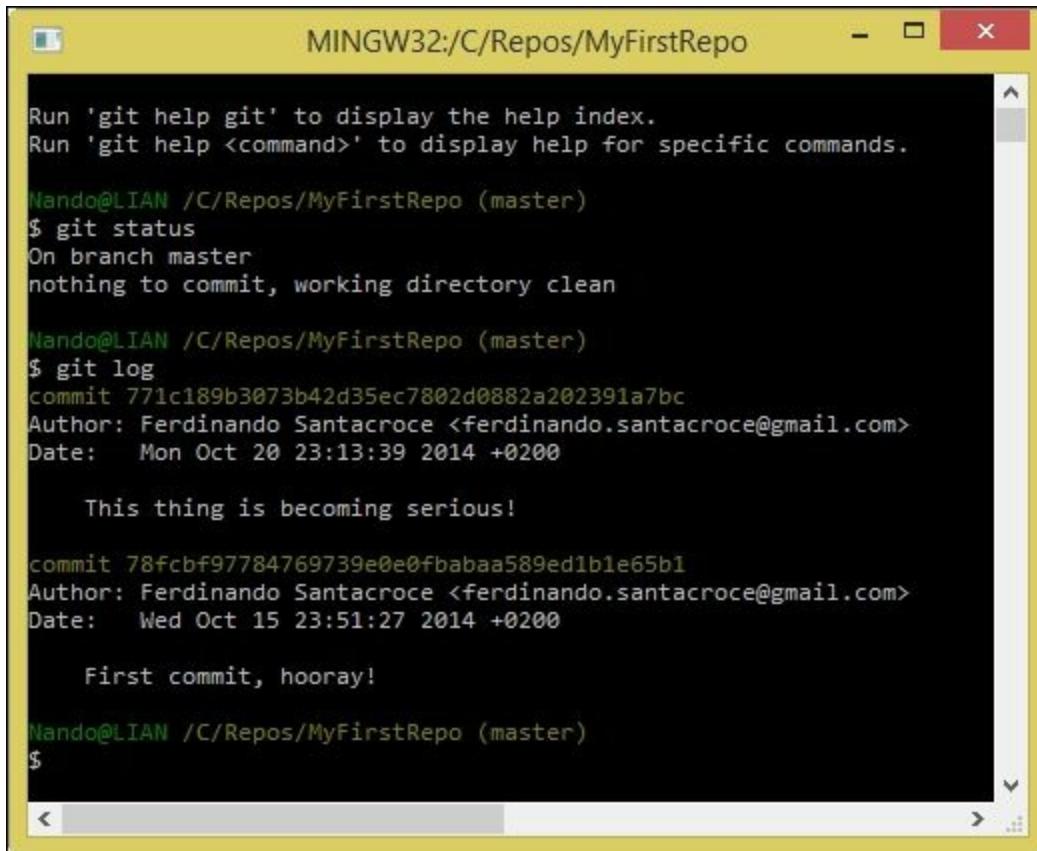
Nando@LIAN /C/Repos/Exercises/Ch1-1 (master)
$ git commit -m "Adds first.txt file"
[master (root-commit) 7b2c279] Adds first.txt file
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 first.txt

Nando@LIAN /C/Repos/Exercises/Ch1-1 (master)
$
```

Well done!

Viewing the history

Let's continue where we left off in [Chapter 1](#), *Getting Started with Git*. Walk into `C:\Repos\MyFirstRepo` and start a new Bash shell using the right-click shortcut. Now, use the `git log` command to see the history of our repository, as shown in this screenshot:



The screenshot shows a terminal window titled "MINGW32:/C/Repos/MyFirstRepo". The window contains the following text output from the git log command:

```
Run 'git help git' to display the help index.
Run 'git help <command>' to display help for specific commands.

Nando@LIAN /C/Repos/MyFirstRepo (master)
$ git status
On branch master
nothing to commit, working directory clean

Nando@LIAN /C/Repos/MyFirstRepo (master)
$ git log
commit 771c189b3073b42d35ec7802d0882a202391a7bc
Author: Ferdinando Santacroce <ferdinando.santacroce@gmail.com>
Date:   Mon Oct 20 23:13:39 2014 +0200

    This thing is becoming serious!

commit 78fcfb97784769739e0e0fbabaa589ed1b1e65b1
Author: Ferdinando Santacroce <ferdinando.santacroce@gmail.com>
Date:   Wed Oct 15 23:51:27 2014 +0200

    First commit, hooray!

Nando@LIAN /C/Repos/MyFirstRepo (master)
$
```

The `git log` command is very useful and powerful. With this command, we can get all the commits that we did one by one, each one with its most important details. It's now time to become more familiar with them.

Anatomy of a commit

Commits are the building blocks of a repository. Every repository is nothing more than an ordered sequence of commits. If you have a math background, you have probably already identified an acyclic direct graph in it.

The commit snapshot

Every time we commit something, Git wraps all the files included in a binary blob file. This is one of the most important characteristics of Git. While other versioning systems save files one by one (perhaps using deltas), Git produces a snapshot every time you commit, storing all the files you have included in it. You can assume that a snapshot is a commit, and vice versa.

The commit hash

The commit hash is the 40-character string we saw in logs. This string is the plate of the commit, the way we can refer to it unequivocally. We will not fail to use it in our little exercises.

Author, e-mail, and date

Git is for collaborating, too. So, it is important that every author signs every commit it does to make it clear who did what.

In every commit, you will find the author's friendly name and their e-mail to get in touch with them when necessary. If you are not happy with the actual author you see, change it with the `git config` command, as shown here:



```
Nando@LIAN /C/Repos/MyFirstRepo (master)
$ git config user.name "Nando"

Nando@LIAN /C/Repos/MyFirstRepo (master)
$ git config user.name
Nando

Nando@LIAN /C/Repos/MyFirstRepo (master)
$
```

Configuring Git is quite simple. You have to know the name of the object to configure (`user.name` in this example) and give it a `git config`

`<object> "<value>"` command. To change the e-mail, you have to change the `user.email` configuration object.

If you want to see the value of an object, simply type `git config <object>` and press *Enter* to get it.

There are three configuration levels: global, user, and repository. We will soon deal with Git configuration in more detail.

Commit messages

In the first chapter, we made two commits, and every time, we added a commit message. Adding a commit message is not only a good practice, it is mandatory. Git would not accept commits without a relative commit message. This is a thing I really appreciate. Let me explain it briefly.

In other versioning control systems, such as SVN, commit messages are optional. Developers, we all know, are lazy people. When they are in a hurry or under stress, the temptation to commit code without telling what feature of the software they are going to add, modify, improve or delete is strong. In no time, you build a mute repository with no change history, and you have to deal with commits that are difficult to understand. In other words, welcome to hell.

Committing a bunch of files

At this point, probably, you are wondering if there is a way to add more than one file at a time. Of course there is! With `git add --all (-A)`, you add all of the files you have in your working directory to the index. You can also use `<filepattern>` to add only certain types of files; for example, with `git add *.txt`, you can add all text files to the index.

Ignoring some files and folders by default

Often, we work with temp or personal files that we don't want to commit in the repository. So, when you commit all files, it is useful to skip certain kinds of files or folders.

To achieve this result, we can create a `.gitignore` file in the repository. Git will read it and then skip the files and folders we listed inside it.

Let's try to do this in our repository, `C:\Repos\MyFirstRepo`. Perform the following steps:

1. Browse to `C:\Repos\MyFirstRepo`.
2. Create a `.gitignore` file using your preferred editor.
3. Put this text inside it:

```
# === This is a sample of .gitignore file ===  
# Ignore temp files  
*.tmp
```

4. Save the file.
5. Add the file to the index.
6. Commit the `.gitignore` file.
7. Create a temp file `fileToIgnore.tmp` with a simple `touch` command.
8. Try to add all of the files in your working directory to the index and verify that Git will not add anything.

```
MINGW32:/C/Repos/MyFirstRepo
Nando@LIAN /C/Repos/MyFirstRepo (master)
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    .gitignore

nothing added to commit but untracked files present (use "git add" to track)

Nando@LIAN /C/Repos/MyFirstRepo (master)
$ git add .gitignore

Nando@LIAN /C/Repos/MyFirstRepo (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   .gitignore

Nando@LIAN /C/Repos/MyFirstRepo (master)
$ git commit -m "Includes the .gitignore file to ignore *.tmp files"
[master 986a6dc] Includes the .gitignore file to ignore *.tmp files
  1 file changed, 4 insertions(+)
  create mode 100644 .gitignore

Nando@LIAN /C/Repos/MyFirstRepo (master)
$ touch fileToIgnore.tmp

Nando@LIAN /C/Repos/MyFirstRepo (master)
$ git add -A

Nando@LIAN /C/Repos/MyFirstRepo (master)
$ git status
On branch master
nothing to commit, working directory clean

Nando@LIAN /C/Repos/MyFirstRepo (master)
$
```

Note that `.gitignore` file is not retroactive. If you have added some `*.tmp` files to the index before introducing the `.gitignore` file, they will stay under revision control. You have to remove them manually if you want to skip them.

The syntax of a `.gitignore` file is quite simple. Lines starting with `#` are comments, and they will be ignored. In each line of the file, you can add something to skip. You can skip a single file or folder, certain files by extension, as we did with `*.tmp` files, and so on.

For the complete syntax of `.gitignore` see <http://git-scm.com/docs/gitignore>.

Tip

To add a file in the `.gitconfig` file even if it is marked to be ignored, you can use the `git add -f (--force)` option.

Highlighting an important commit – Git tags

We said that commits have an ID and a lot of other useful information bundled with binary blobs of files included. Sometimes, we want to mark a commit with a tag to place a milestone in our repository. You can achieve this by simply using the `git tag -a <tag name>` command, using `-m` to type the mandatory message:

```
$ git tag -a MyTagName -m "This is my first tag"
```

Tags will become useful in the future to keep track of important things such as a new software release, particular bug fixes, or whatever you want to put on evidence.

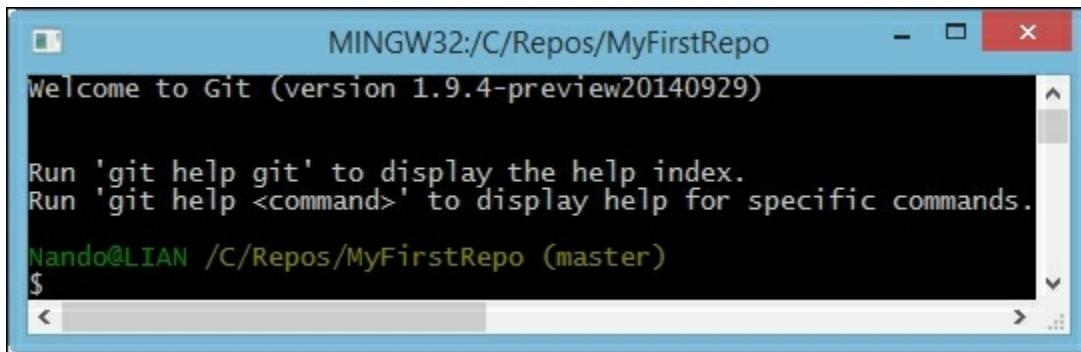
Taking another way – Git branching

Now, it's time to introduce one of the most used features of a versioning system: **branching**. This is a thing that you will use extensively and a thing that Git does well.

Anatomy of branches

A branch is essentially another way your repository takes. While programming, you will use branches to experiment with changes, without breaking the working code. You will use them to keep track of some extensive work, such as the development of a new feature, to maintain different versions or releases of your project. To put it simply, you will use it always.

When in a Git repository, you are always in a branch. If you pay attention to the Bash shell, you can easily find the branch that you are on at the moment. It is always at the prompt, within brackets, as shown here:



The screenshot shows a terminal window titled "MINGW32:/C/Repos/MyFirstRepo". The window displays the following text:
Welcome to Git (version 1.9.4-preview20140929)
Run 'git help git' to display the help index.
Run 'git help <command>' to display help for specific commands.
Nando@LIAN /C/Repos/MyFirstRepo (master)
\$

The `master` branch is, conventionally, the default branch. When you do your first commit in a brand new repo, you actually do the first commit in the `master` branch.

During the course of the book, you will learn that the `master` branch will

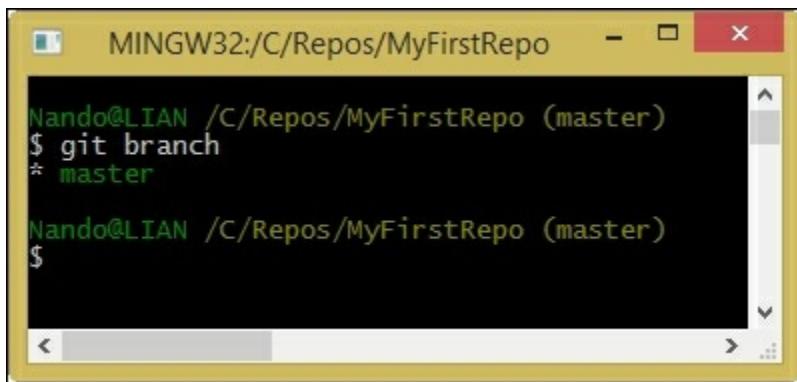
often assume an important role. We will only deploy code from that point, and we will never work directly in the `master` branch.

Looking at the current branches

Let's start using the `git branch` command alone. If you do this, you will get a list of the branches included in the current repository:

```
$ git branch
```

We will get the following response:



The screenshot shows a terminal window titled "MINGW32:/C/Repos/MyFirstRepo". The window contains the following text:

```
Nando@LIAN /C/Repos/MyFirstRepo (master)
$ git branch
* master

Nando@LIAN /C/Repos/MyFirstRepo (master)
$
```

As expected, Git tells us that we have only the `master` branch. The branch is highlighted with a star and green font just because this is our actual branch, the branch in which we are located at the moment.

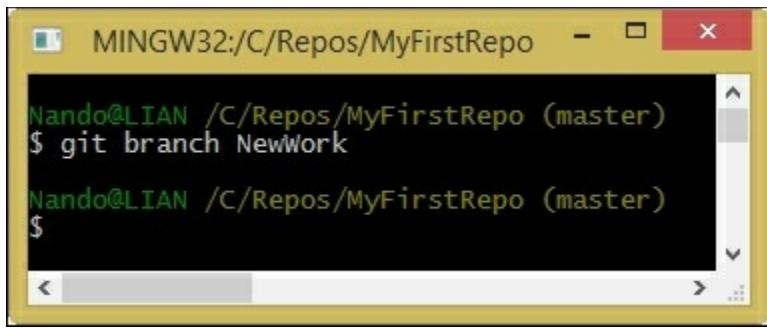
Creating a new branch

Now, let's start creating a new branch for a new activity.

Open the Bash shell in `C:\Repos\MyFirstRepo` and create a new branch from where you are using the `git branch` command, followed by the name of the branch we want to create. Let's assume that the name of the branch is `NewWork`:

```
$ git branch NewWork
```

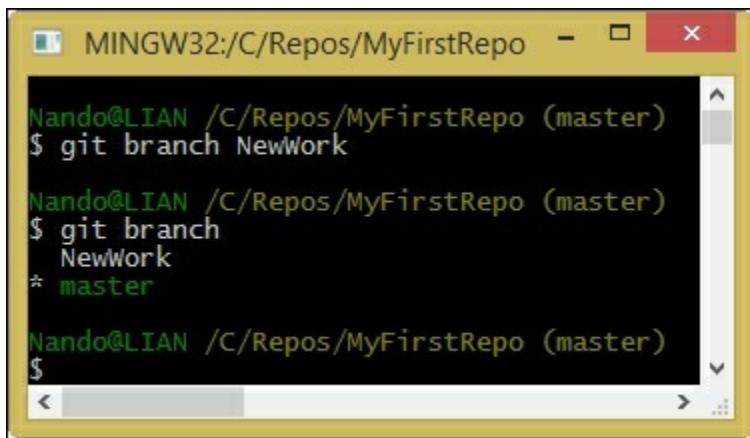
Well, it seems nothing happened and we are still in the `master` branch as we can see in the following screenshot:



A screenshot of a terminal window titled "MINGW32:/C/Repos/MyFirstRepo". The window shows the following command sequence:

```
Nando@LIAN /C/Repos/MyFirstRepo (master)
$ git branch NewWork
Nando@LIAN /C/Repos/MyFirstRepo (master)
$
```

Git sometimes is a man of few words. Type the `git branch` command again to get a list of the actual branches:



A screenshot of a terminal window titled "MINGW32:/C/Repos/MyFirstRepo". The window shows the following command sequence:

```
Nando@LIAN /C/Repos/MyFirstRepo (master)
$ git branch NewWork
Nando@LIAN /C/Repos/MyFirstRepo (master)
$ git branch
  NewWork
* master
Nando@LIAN /C/Repos/MyFirstRepo (master)
$
```

Hey, there's a `NewWork` branch now!

Switching from branch to branch

The only thing we have to do now is to switch to the `NewWork` branch to begin working on this separate argument. To move around from one branch to another, we will use the `git checkout` command. Type this command followed by the branch name to switch to:

```
$ git checkout NewWork
```

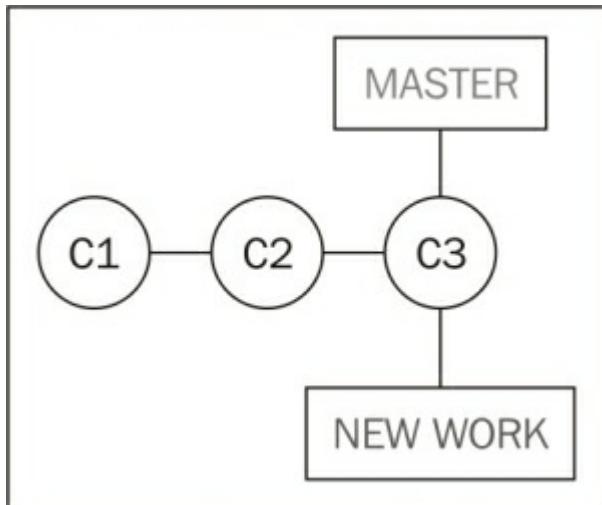
This time, Git gave us a short message, informing us that we have switched to the `NewWork` branch. As you can see, `NewWork` is within brackets in place of `master`, indicating the actual branch to us:

```
MINGW32:/C/Repos/MyFirstRepo - □ X  
Nando@LIAN /C/Repos/MyFirstRepo (master)  
$ git branch NewWork  
Nando@LIAN /C/Repos/MyFirstRepo (master)  
$ git branch  
  NewWork  
* master  
Nando@LIAN /C/Repos/MyFirstRepo (master)  
$ git checkout NewWork  
Switched to branch 'NewWork'  
Nando@LIAN /C/Repos/MyFirstRepo (NewWork)  
$
```

Super easy, isn't it?

Understanding what happens under the hood

At this point, nothing special seems to have happened. However, you will soon change your mind. Just to give you a visual aid, consider this figure:



Assume that our last commit on the `master` branch was the **C3** commit. At this point, we now have two other pointers: the `master` branch pointer and the `NewWork` branch pointer.

If you have used Subversion or a similar versioning system earlier, at this point, you would probably look for a `NewWork` folder in `C:\Repos\MyFirstRepo`. However, unfortunately, you will not find it. Git is different, and now, we will see why.

Now, to better understand what a branch is for, add `NewWorkFile.txt`, stage it, and commit the work, as shown here:

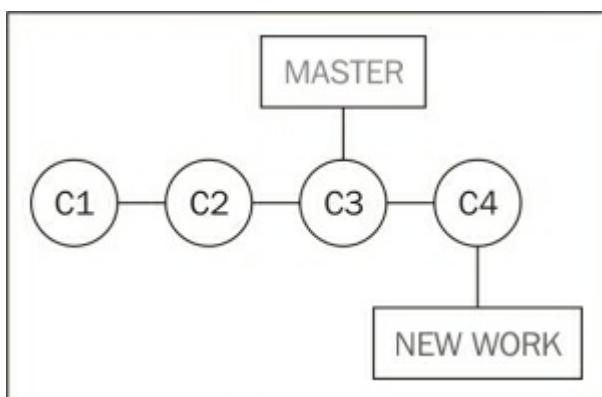
```
MINGW32:/C/Repos/MyFirstRepo
Nando@LIAN /C/Repos/MyFirstRepo (NewWork)
$ echo "This new file has been created on NewWork branch" >> NewWorkFile.txt

Nando@LIAN /C/Repos/MyFirstRepo (NewWork)
$ git add NewWorkFile.txt
warning: LF will be replaced by CRLF in NewWorkFile.txt.
The file will have its original line endings in your working directory.

Nando@LIAN /C/Repos/MyFirstRepo (NewWork)
$ git commit -m "This commit has been done to the NewWork branch"
[NewWork a0c7d61] This commit has been done to the NewWork branch
warning: LF will be replaced by CRLF in NewWorkFile.txt.
The file will have its original line endings in your working directory.
1 file changed, 1 insertion(+)
create mode 100644 NewWorkFile.txt

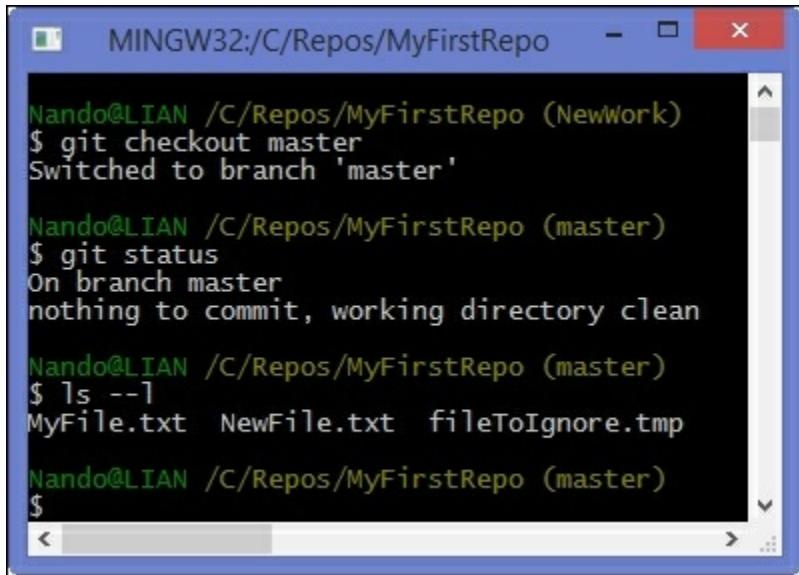
Nando@LIAN /C/Repos/MyFirstRepo (NewWork)
$
```

At this point, we have a new commit in the `NewWork` branch. This commit is not a part of the `master` branch, because it has been created in another branch. So, the `NewWork` branch is ahead of the `master` branch, as shown in the following figure:



Working with Git in this kind of a situation is ordinary administration, just like it is in the day-to-day life of a developer. Creating a new branch is cheap and blazing fast even in large repositories, because all the work is done locally. You can derive multiple branches from a unique point, branch from a branch that branched into another branch, and so on, creating all the ramifications you can manage without going crazy.

Moving from a branch to another is easy. So, let's turn back to the master branch and see what happens:



```
Nando@LIAN /C/Repos/MyFirstRepo (NewWork)
$ git checkout master
Switched to branch 'master'

Nando@LIAN /C/Repos/MyFirstRepo (master)
$ git status
On branch master
nothing to commit, working directory clean

Nando@LIAN /C/Repos/MyFirstRepo (master)
$ ls -l
MyFile.txt  NewFile.txt  fileToIgnore.tmp

Nando@LIAN /C/Repos/MyFirstRepo (master)
$
```

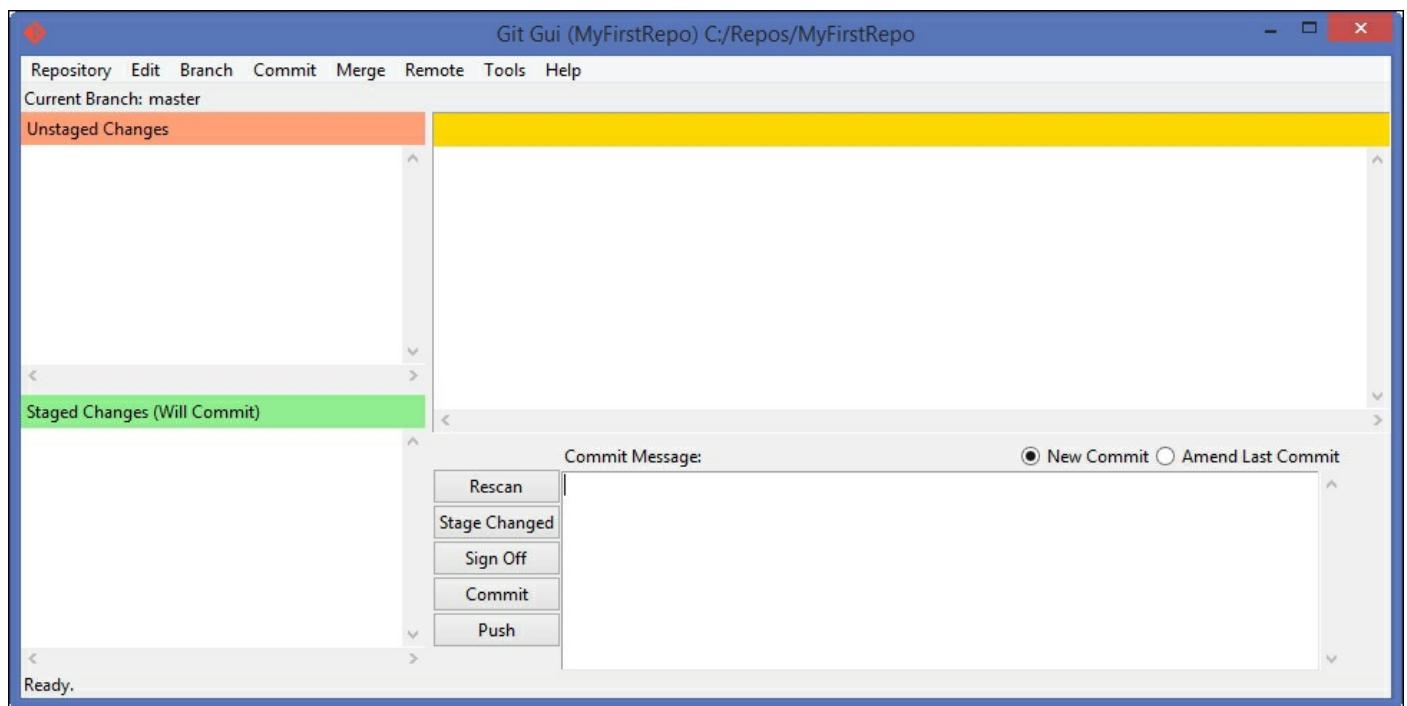
If you have used Subversion earlier, at this time, you are probably wondering at least two things: why is there no `NewWork` folder containing files from that branch and, most of all, what was the fate of the `NewWorkFile.txt` file.

In Apache Subversion (SVN), every branch (or tag) you checked out resides in a separate folder (bloating your repository, after some time). In Git, there is a huge difference. Every time you check out a branch, files and folders of the previous branch are replaced with those contained in the new branch. Files and folders only on the destination branch are restored, and those only on the previous branch are deleted. Your working directory is constantly a mirror of the actual branch. To have a look at another branch, you basically have to switch to it.

This aspect might look bad to you the first time, because seeing files and folders disappear is scary at first. Then, you can't differentiate the two branches as we probably did earlier by simply comparing the content of the two folders with your preferred tool. If I can give you a piece of advice, don't lose heart about this (as I did at the beginning): soon you will forget what you lost and you will fall in love with what you gained.

A bird's eye view to branches

So, let's go back once again to the `NewWork` branch and have a look at the situation with the aid of a visual tool, Git GUI. Open the `C:\Repos\MyFirstRepo` folder and right-click inside it. Choose **Git GUI** from the contextual menu. A dialog box will pop up as shown in the following screenshot:



Git GUI is not my preferred GUI tool, but you have it for free when installing Git. So, let's use it for the moment.

Go to **Repository | Visualize All Branch History**. A new window will open, and you will see the status of our current branches, as shown in the following screenshot:

git: MyFirstRepo

File Edit View Help

NewWork This commit has been done to the NewWork branch
 master Includes the .gitignore file to ignore *.tmp files
 Adding the new file
 This thing is becoming serious!
 First commit, hooray!

Ferdinando Santacroce <ferdinando.santacroce@gmail.com> 2014-11-22 17:49:16
 Ferdinand Santacroce <ferdinando.santacroce@gmail.com> 2014-11-19 07:25:02
 Ferdinand Santacroce <ferdinando.santacroce@gmail.com> 2014-11-19 07:22:54
 Ferdinand Santacroce <ferdinando.santacroce@gmail.com> 2014-10-20 23:13:39
 Ferdinand Santacroce <ferdinando.santacroce@gmail.com> 2014-10-15 23:51:27

SHA1 ID: 986a6dc3dff25e378c2d614cfe413a44b524980f | Row | 2 / 5 |

Find ↓ ↑ commit containing: | Exact | All fields |

Search | Patch | Tree | Comments | .gitignore

Diff Old version New version Lines of context: 3 | Ignore space change |

Author: Ferdinando Santacroce <ferdinando.santacroce@gmail.com> 2014-11-19 07:25:02
 Committer: Ferdinando Santacroce <ferdinando.santacroce@gmail.com> 2014-11-19 07:25:02
 Parent: 3e5931b2ff5dea314dd252ddd35da88d7bbac73 (Adding the new file)
 Child: a0c7d618cf8a2f71b2da8e75313fb800149089 (This commit has been done to the NewWork branch)
 Branches: NewWork, master
 Follows:
 Precedes:

Includes the .gitignore file to ignore *.tmp files

```
----- .gitignore -----
new file mode 100644
index 0000000..66a3105
@@ -0,0 +1,4 @@
+## === This is a sample of .gitignore file ===
+
+## Ignore temp files
+**.tmp
\ No newline at end of file
```

We do not have the time to go into all the details here. As you become more confident with Git fundamentals, you will learn little by little all the things you see in the preceding picture.

If you don't want to leave the console to take a look, you could get a pretty output log even on the console. Try this articulated `git log` command:

```
$git log --graph --decorate --pretty=oneline --abbrev-commit
```

MINGW32:/C/Repos/MyFirstRepo

```
Nando@LIAN /C/Repos/MyFirstRepo (NewWork)
$ git log --graph --decorate --pretty=oneline --abbrev-commit
* a0c7d61 (HEAD, NewWork) This commit has been done to the NewWork branch
* 986a6dc (master) Includes the .gitignore file to ignore *.tmp files
* 3e5931b Adding the new file
* 771c189 This thing is becoming serious!
* 78fcfb9 First commit, hooray!
```

Nando@LIAN /C/Repos/MyFirstRepo (NewWork)

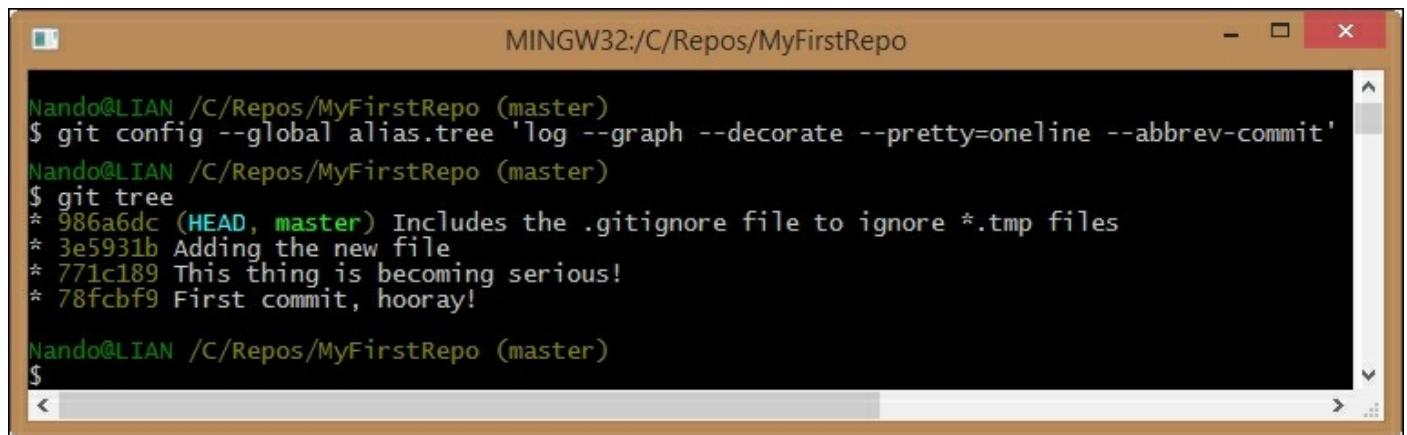
This is a more compact view, but it is as clear as what we saw earlier.

Typing is boring – Git aliases

Before continuing, just a little warning. The last command we used is not the easiest thing to remember. To get around this nuisance, Git offers the possibility of creating your own commands, aliasing some of the verbose sequences.

To create an alias, we will use the `git config` command. So, let's try to create a `tree` alias for this command:

```
$ git config --global alias.tree 'log --graph --decorate --pretty=oneline --abbrev-commit'
```



The screenshot shows a terminal window with the title 'MINGW32:/C/Repos/MyFirstRepo'. The command \$ git config --global alias.tree 'log --graph --decorate --pretty=oneline --abbrev-commit' is entered and executed. Then, the command \$ git tree is entered and executed, displaying a log of commits. The commits shown are:

- * 986a6dc (HEAD, master) Includes the .gitignore file to ignore *.tmp files
- * 3e5931b Adding the new file
- * 771c189 This thing is becoming serious!
- * 78fcfb9 First commit, hooray!

So, the syntax to create aliases is as follows:

```
git config <level> alias.<alias name> '<your sequence of git commands>'
```

Using the `--global` option, we told Git to insert this alias at the user level so that any other repository for my user account on this computer will have this command available. We have three levels where we can apply config personalization:

- Repository level configs are only available for the current repo
- Global configs are available for all the repos for the current user
- System configs are available for all the users/repositories

To create a repository user-specific config, we used the `--global` option. For the system level, we will use the `--system` option. If you don't specify any of these two options, the config will take effect only in the repository that you are in now.

Merging branches

Now that we finally got in touch with branches, let's assume that our work in the `NewWork` branch is done and we want to bring it back to the `master` branch.

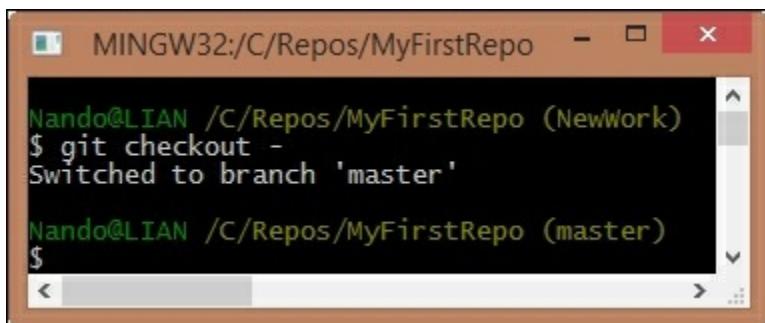
To merge two branches, we have to move to the branch that contains the other branch commits. So, if we want to merge the `NewWork` branch into the `master` branch, we would first have to check out the `master` branch. As seen earlier, to check out a branch we have to type the following command:

```
$ git checkout <branch name>;
```

If you want to check out the previous branch you were in, it's even simpler. Type this command:

```
$ git checkout -
```

With the `git checkout -` command, you will move in the previous branch without having to type its name again as explained in the following screenshot:

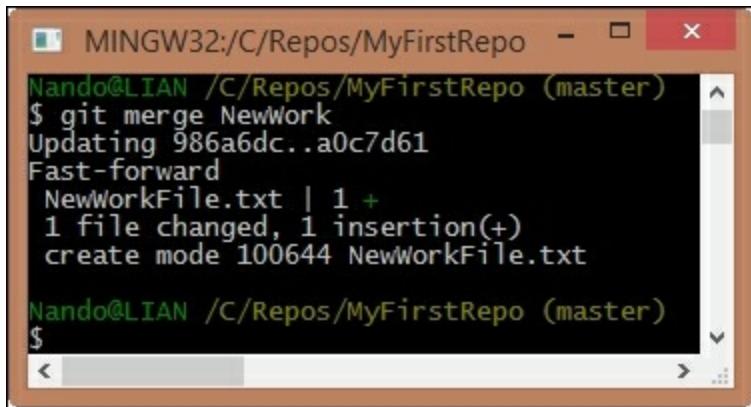


The screenshot shows a terminal window titled "MINGW32:/C/Repos/MyFirstRepo". The command \$ git checkout - is entered, followed by the output "Switched to branch 'master'". The prompt then changes to show the master branch: Nando@LIAN /C/Repos/MyFirstRepo (master)

Now, let's merge the `NewWork` branch into `master` using the `git merge` command:

```
$ git merge NewWork
```

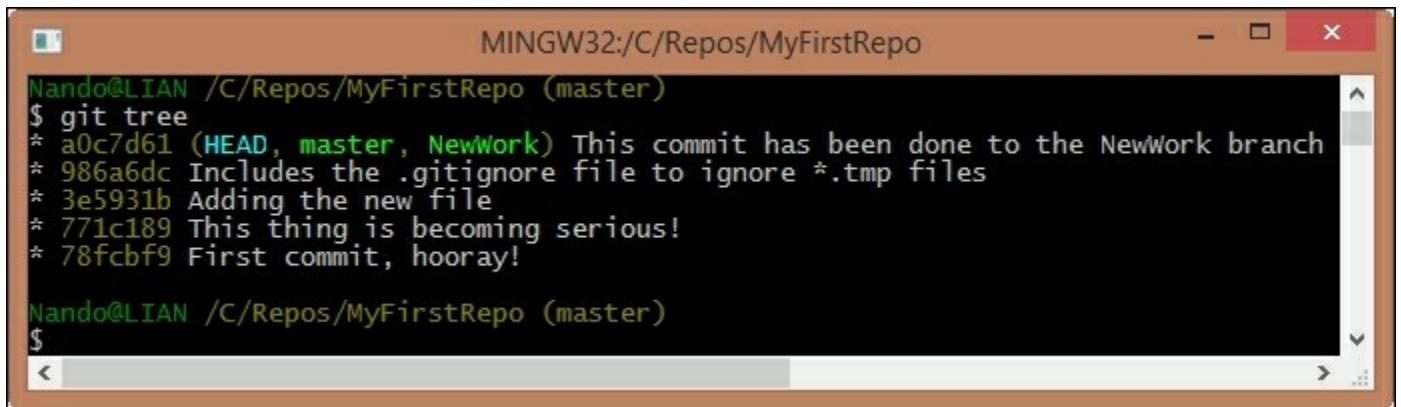
As you can see, merging is quite simple. With the `git merge <branch to merge>` command, you can merge all the modifications in a branch into the actual branch as shown in the following screenshot:



A screenshot of a terminal window titled "MINGW32:/C/Repos/MyFirstRepo". The command \$ git merge NewWork is run, resulting in the following output:
Updating 986a6dc..a0c7d61
Fast-forward
NewWorkFile.txt | 1 +
1 file changed, 1 insertion(+)
create mode 100644 NewWorkFile.txt

When we read console messages, we see that Git is telling us something interesting. Let's take a look.

When Git says Updating 986a6dc..a0c7d61, it is telling us that it is updating pointers. So now, we will have master, NewWork, and HEAD all pointing to the same commit:



A screenshot of a terminal window titled "MINGW32:/C/Repos/MyFirstRepo". The command \$ git tree is run, resulting in the following output:
* a0c7d61 (HEAD, master, NewWork) This commit has been done to the NewWork branch
* 986a6dc Includes the .gitignore file to ignore *.tmp files
* 3e5931b Adding the new file
* 771c189 This thing is becoming serious!
* 78fcfb9 First commit, hooray!

Fast-forward is a concept that we will cover soon. When we tell Git to not apply this feature, in brief, this fast-forward feature (enabled by default) permits us to merge commits from different branches, as they were done subsequently in the same branch, obtaining a less-pronged

repository.

Last, but not least, we have a complete report of what files are added, deleted, or modified. In our case, we had just one change (an insertion) and nothing else. When something is added, Git uses a green plus symbol +. When a deletion happens, Git uses a red dash symbol -.

Merge is not the end of the branch

This is a concept that sometimes raises some trouble. To avoid this, we will have a quick glimpse of it. You don't have to wait for the work to be done before merging another branch into your branch. Similarly, you don't have to think about merging as the last thing you will do before cutting off the other branch on which you worked.

On the contrary, it's better if you merge frequently from branches you depend on, because doing it after weeks or months can become a nightmare: too many changes in the same files, too many additions or deletions. Don't make a habit of this!

Exercises

To understand some common merging scenarios, you can try to resolve these small exercises.

Exercise 2.1

In this exercise we will learn how Git can handle automatically file modifications when they are not related to the same lines of text.

What you will learn

Git is able to automerge modifications on the same files.

Scenario

1. You are in a new repository located in C:\Repos\Exercises\Ch2-1.
2. You have a `master` branch with two previous commits: the first commit with a `file1.txt` file and the second commit with a `file2.txt` file.

3. After the second commit, you created a new branch called `File2Split`. You realized that `file2.txt` is too big, and you want to split its content by creating a new `file2a.txt` file. Do it, and then commit the modifications.

Results

Merging back `File2Split` in the `master` branch is a piece of cake. Git takes care of deletion in `file2.txt` without spotting your conflicts.

Exercise 2.2

In this exercise we will learn how to resolve conflicts when Git cannot merge files automatically.

What you will learn

Git will result in conflicts if automerging is not possible.

Scenario

1. You are in the same repository used earlier,
`C:\Repos\Exercises\Ch2-1`.
2. On the `master` branch, you add the `file3.txt` file and commit it.
3. Then, you realize that it is better to create a new branch to work on `file3.txt`, so you create the `File3Work` branch. You move in this branch, and you start to work on it, committing modifications.
4. The day after, you accidentally move to the `master` branch and make some modifications on the `file3.txt` file, committing it.
5. Then, you try to merge it.

Results

Merging the `File3Work` branch in `master` gives rise to some conflicts. You have to solve them manually and then commit the merged modifications.

Deal with branches' modifications

As mentioned earlier, if you come from SVN at this point, you would be

a little confused. You don't "physically have on the disk" all the branches checked out on different folders. Because of this, you cannot easily differentiate two branches to take into account what a merge will cost in terms of conflicts to resolve.

Well, for the first problem, there is not an SVN-like solution. However, if you really want to differentiate two checked out branches, you could copy the working directory in a `temp` folder and check out the other branch. This is just a workaround, but the first time can be a less traumatic way to manage the mental shift Git applies in this field.

Diffing branches

If you want to do it in a more Git-like way, you could use the `git diff` command. Let's give it a try by performing the following steps:

1. Open `C:\Repos\MyFirstRepo` and switch to the `master` branch.
2. Add some text to the existing `NewFile.txt`, and then save it.
3. Add a `NewMasterFiles.txt` file with some text within it. At the end, add both files to the index and then commit them, as shown in the following screenshot:

The screenshot shows a terminal window with the following command history:

```
Nando@LIAN /C/Repos/MyFirstRepo (master)
$ echo "This is a new file on master branch" >> NewMasterFile.txt

Nando@LIAN /C/Repos/MyFirstRepo (master)
$ notepad NewFile.txt

Nando@LIAN /C/Repos/MyFirstRepo (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   NewFile.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    NewMasterFile.txt

no changes added to commit (use "git add" and/or "git commit -a")

Nando@LIAN /C/Repos/MyFirstRepo (master)
$ git add NewFile.txt NewMasterFile.txt
Nando@LIAN /C/Repos/MyFirstRepo (master)
$ git commit -m 'Adds a NewMasterFile.txt and adds text to NewFile.txt'
[master cd154fa] Adds a NewMasterFile.txt and adds text to NewFile.txt
 2 files changed, 2 insertions(+)
 create mode 100644 NewMasterFile.txt

Nando@LIAN /C/Repos/MyFirstRepo (master)
```

Now, try to have a look at the differences between the `master` and `NewWork` repositories. Finding out the difference between the two branches is easy with the `git diff` command:

```
$ git diff master..NewWork
```

The syntax is simple: `git diff <source branch>..<target branch>`. The result, instead, is not probably the clearest thing you have ever seen:

```
Nando@LIAN /C/Repos/MyFirstRepo (master)
$ git diff master..NewWork
diff --git a/NewFile.txt b/NewFile.txt
index 0ae6d47..e69de29 100644
--- a/NewFile.txt
+++ b/NewFile.txt
@@ -1 +0,0 @@
-Added some text to the existing NewFile
\ No newline at end of file
diff --git a/NewMasterFile.txt b/NewMasterFile.txt
deleted file mode 100644
index 679667b..0000000
--- a/NewMasterFile.txt
+++ /dev/null
@@ -1 +0,0 @@
+This is a new file on master branch
```

However, with a little imagination, you can understand that the differences are described from the point of view of the `NewWork` branch. Git is telling us that some things on `master` (`NewFile.txt` modifications and `NewMasterFile.txt`) are not present in the `NewWork` branch.

If we change the point of view, the messages change to those shown in the following screenshot:

```
Nando@LIAN /C/Repos/MyFirstRepo (master)
$ git diff NewWork..master
diff --git a/NewFile.txt b/NewFile.txt
index e69de29..0ae6d47 100644
--- a/NewFile.txt
+++ b/NewFile.txt
@@ -0,0 +1 @@
+Added some text to the existing NewFile
\ No newline at end of file
diff --git a/NewMasterFile.txt b/NewMasterFile.txt
new file mode 100644
index 0000000..679667b
--- /dev/null
+++ b/NewMasterFile.txt
@@ -0,0 +1 @@
+This is a new file on master branch
```

Note

To better understand these messages, you can take a look at the diff output at http://en.wikipedia.org/wiki/Diff_utility.

Another way to check differences is to use `git log`:

```
$ git log NewWork..master
```

This command lets you see commits that differ from the `NewWork` branch to the `master` branch.

There is even a `git shortlog` command to give you a more compact view, as shown in the following screenshot:

```
Nando@LIAN /C/Repos/MyFirstRepo (master)
$ git log NewWork...master
commit cd154fae66f188c33b2f5e48b68dead989ed7187
Author: Ferdinando Santacroce <ferdinando.santacroce@gmail.com>
Date:   Sun Nov 23 18:47:39 2014 +0100

    Adds a NewMasterFile.txt and ads text to NewFile.txt

Nando@LIAN /C/Repos/MyFirstRepo (master)
$ git shortlog NewWork...master
Ferdinando Santacroce (1):
    Adds a NewMasterFile.txt and ads text to NewFile.txt
```

Using a visual diff tool

All these commands are useful for short change history. However, if you have a more long change list to scroll, things would quickly become complicated.

Git lets you use an external diff tool of choice. On other platforms (for example, Linux or Mac), a diff tool is usually present and configured, while on the Windows platform, it is generally not present.

To check this, type this command:

```
$ git mergetool
```

If you see a message like this, you would probably have to set your preferred tool:

```
Nando@LIAN /C/Repos/MyFirstRepo (master)
$ git mergetool

This message is displayed because 'merge.tool' is not configured.
See 'git mergetool --tool-help' or 'git help config' for more details.
'git mergetool' will now attempt to use one of the following tools:
tortoisemerge emerge vimdiff
No known merge tool is available.
```

Resolving merge conflicts

As we have seen, merging branches is not a difficult task. However, in real-life scenarios, things are not that easy. We have conflicts, modifications on both branches, and other weird things to fight. In this section, we will take a look at some of them. However, first, remember one important thing: it needs a little bit of discipline to make the most of Git.

You have to avoid at least two things:

- Working hard on the same files on different branches
- Rarely merging branches

Edit collisions

This is the most common kind of conflict: someone edited the same line in the same file on different branches, so Git can't auto merge them for you. When this happens, Git writes special conflict markers to the affected areas of the file. At this point, we have to manually solve the situation, editing that area to fit our needs.

Let's try this by performing the following steps:

1. Open your repository located in `C:\Repos\MyRepos`.
2. Switch to the `NewWork` branch and edit `NewFile.txt` by modifying the first line `Added some text to the existing NewFile in Text has been modified.`
3. Add and commit the modification.
4. Switch back to `master` and merge the `NewWork` branch.

```
MINGW32:/C/Repos/MyFirstRepo
Nando@LIAN /C/Repos/MyFirstRepo (NewWork)
$ echo "Text has been modified" > NewFile.txt

Nando@LIAN /C/Repos/MyFirstRepo (NewWork)
$ git add NewFile.txt
warning: LF will be replaced by CRLF in NewFile.txt.
The file will have its original line endings in your working directory.

Nando@LIAN /C/Repos/MyFirstRepo (NewWork)
$ git commit -m "File content has been modified"
[NewWork warning: LF will be replaced by CRLF in NewFile.txt.
The file will have its original line endings in your working directory.
d29a1df] File content has been modified
warning: LF will be replaced by CRLF in NewFile.txt.
The file will have its original line endings in your working directory.
 1 file changed, 1 insertion(+), 1 deletion(-)

Nando@LIAN /C/Repos/MyFirstRepo (NewWork)
$ git checkout -
Switched to branch 'master'

Nando@LIAN /C/Repos/MyFirstRepo (master)
$ git merge NewWork
Auto-merging NewFile.txt
CONFLICT (content): Merge conflict in NewFile.txt
Automatic merge failed; fix conflicts and then commit the result.

Nando@LIAN /C/Repos/MyFirstRepo (master|MERGING)
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

    both modified:  NewFile.txt

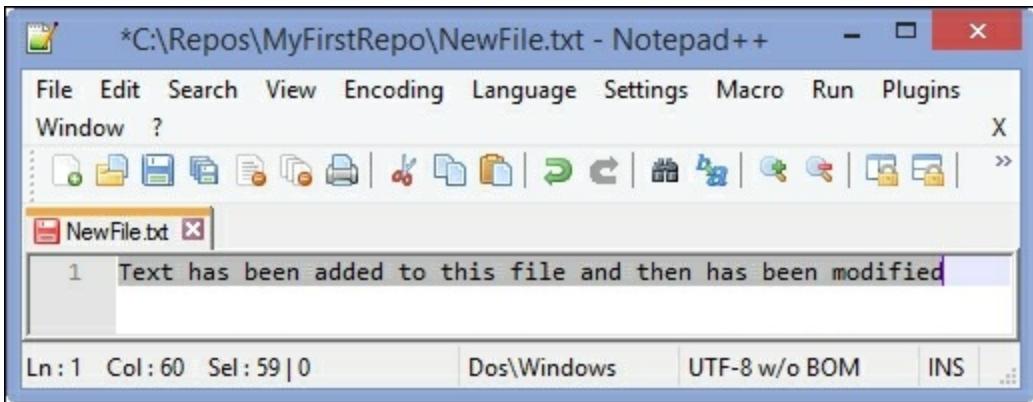
no changes added to commit (use "git add" and/or "git commit -a")

Nando@LIAN /C/Repos/MyFirstRepo (master|MERGING)
$ cat NewFile.txt
<<<<< HEAD
Added some text to the existing NewFile
=====
Text has been modified
>>>>> NewWork

Nando@LIAN /C/Repos/MyFirstRepo (master|MERGING)
$
```

As you can see, Git highlighted the conflict. A conflict-marked area begins with <<<<< and ends with >>>>>. The two conflicting blocks themselves are divided by a sequence of ======. To solve the conflict, you have to manually edit the file, deciding what to maintain, edit, or delete. After that, remove the conflict markers and commit changes to

mark the conflict as resolved.



Once you resolve the conflicts, you are ready to add and commit:

A screenshot of a terminal window titled "MINGW32:/C/Repos/MyFirstRepo". The window displays a command-line session:

```
Nando@LIAN /C/Repos/MyFirstRepo (master|MERGING)
$ git add NewFile.txt

Nando@LIAN /C/Repos/MyFirstRepo (master|MERGING)
$ git commit -m "NewFile.txt conflict has been resolved"
[master 2da22c5] NewFile.txt conflict has been resolved

Nando@LIAN /C/Repos/MyFirstRepo (master)
$ git status
On branch master
nothing to commit, working directory clean

Nando@LIAN /C/Repos/MyFirstRepo (master)
$
```

Merge done, congratulations!

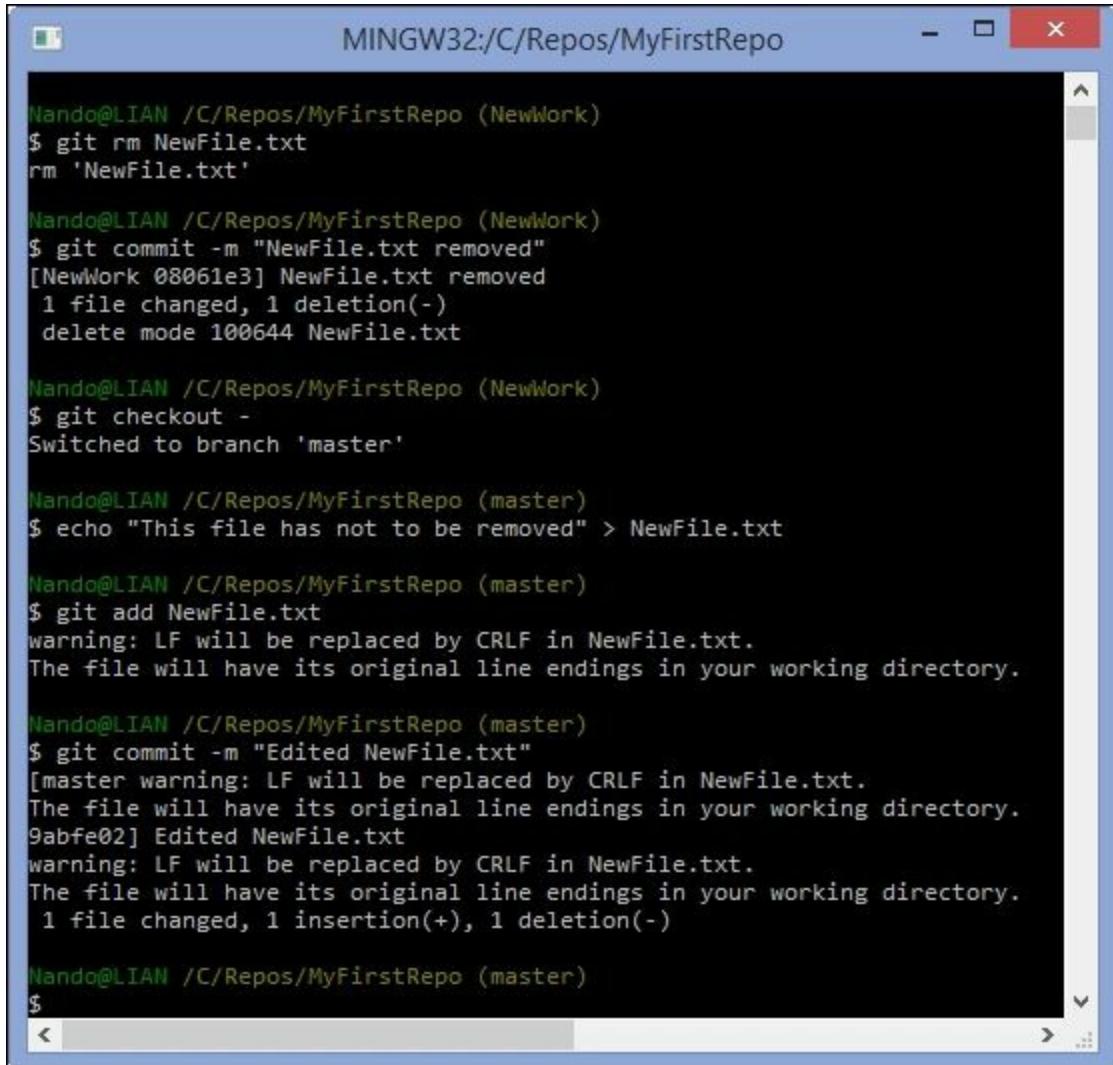
Resolving a removed file conflict

Removed file conflicts occur when you edit a file in a branch and another person deletes that file in their branch. Git does not know if you want to keep the edited file or delete it, so you have to take the decision. This example will show you how to resolve this both ways.

Keeping the edited file

Try again using `NewFile.txt`. Remove it from the `NewWork` branch and

then modify it in master:



```
Nando@LIAN /C/Repos/MyFirstRepo (NewWork)
$ git rm NewFile.txt
rm 'NewFile.txt'

Nando@LIAN /C/Repos/MyFirstRepo (NewWork)
$ git commit -m "NewFile.txt removed"
[NewWork 08061e3] NewFile.txt removed
 1 file changed, 1 deletion(-)
 delete mode 100644 NewFile.txt

Nando@LIAN /C/Repos/MyFirstRepo (NewWork)
$ git checkout -
Switched to branch 'master'

Nando@LIAN /C/Repos/MyFirstRepo (master)
$ echo "This file has not to be removed" > NewFile.txt

Nando@LIAN /C/Repos/MyFirstRepo (master)
$ git add NewFile.txt
warning: LF will be replaced by CRLF in NewFile.txt.
The file will have its original line endings in your working directory.

Nando@LIAN /C/Repos/MyFirstRepo (master)
$ git commit -m "Edited NewFile.txt"
[master warning: LF will be replaced by CRLF in NewFile.txt.
The file will have its original line endings in your working directory.
9abfe02] Edited NewFile.txt
warning: LF will be replaced by CRLF in NewFile.txt.
The file will have its original line endings in your working directory.
 1 file changed, 1 insertion(+), 1 deletion(-)

Nando@LIAN /C/Repos/MyFirstRepo (master)
$
```

After that, merge NewWork in the master branch. As said earlier, Git spots a conflict. Add NewFile.txt again in the master branch and commit it to fix the problem:

```
MINGW32:/C/Repos/MyFirstRepo
Nando@LIAN /C/Repos/MyFirstRepo (master)
$ git merge NewWork
CONFLICT (modify/delete): NewFile.txt deleted in NewWork and modified in HEAD. Version HEAD of NewFile.txt left in tree.
Automatic merge failed; fix conflicts and then commit the result.
Nando@LIAN /C/Repos/MyFirstRepo (master|MERGING)
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add/rm <file>..." as appropriate to mark resolution)

    deleted by them: NewFile.txt

no changes added to commit (use "git add" and/or "git commit -a")

Nando@LIAN /C/Repos/MyFirstRepo (master|MERGING)
$ git add NewFile.txt
warning: LF will be replaced by CRLF in NewFile.txt.
The file will have its original line endings in your working directory.

Nando@LIAN /C/Repos/MyFirstRepo (master|MERGING)
$ git commit
[master 28ee35e] Merge branch 'NewWork'

Nando@LIAN /C/Repos/MyFirstRepo (master)
$
```

When resolving merge conflicts, try to commit without specifying a message, using only the `git commit` command. Git will open the Vim editor, suggesting a default merge message:

```
COMMIT_EDITMSG (c:\Repos\MyFirstRepo\.git) - VIM
Merge branch 'NewWork'

Conflicts:
  NewFile.txt
#
# It looks like you may be committing a merge.
# If this is not correct, please remove the file
#       .git/MERGE_HEAD
# and try again.

#
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# All conflicts fixed but you are still merging.
#
```

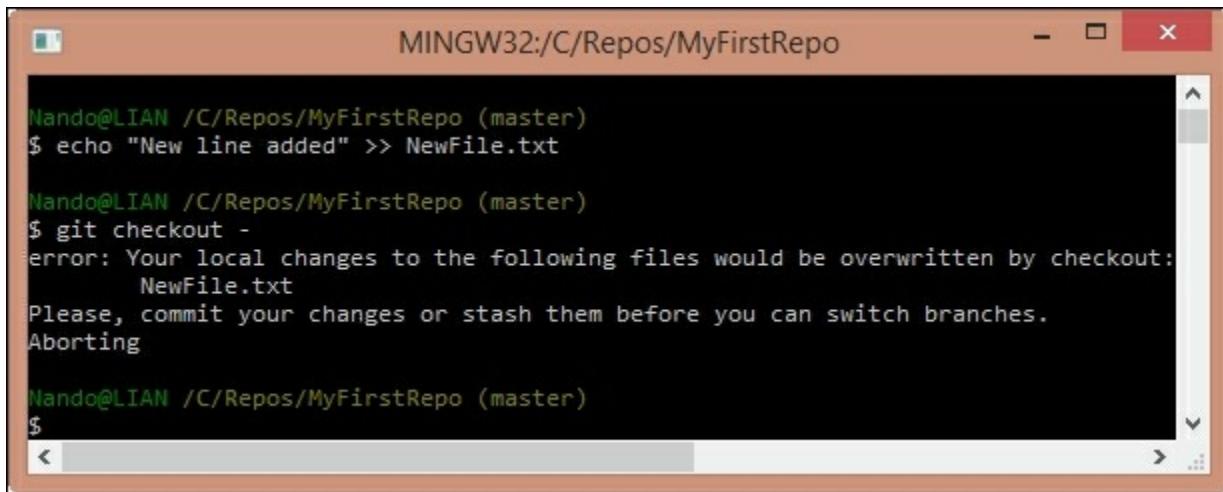
This can be useful to spot merge commits looking at the repository history in the future.

Resolving conflicts by removing the file

If you agree with the deletion of the file in the other branch, instead of adding it again, remove it from your branch. So, use the `git rm NewFile.txt` command even in the `master` branch and then commit to mark the conflict resolved.

Stashing

Working of different features in parallel does not make a developer happy, but sometimes it happens. So, at a certain point, we have to break the work on a branch and switch to another one. However, sometimes, we have some modifications that are not ready to be committed, because they are partial, inconsistent, or even won't compile. In this situation, Git prevents you from switching to another branch. You can only switch from one branch to another if you are in a clean state:



The screenshot shows a terminal window titled "MINGW32:/C/Repos/MyFirstRepo". The user has run the command \$ echo "New line added" >> NewFile.txt. Then, they attempt to switch branches with \$ git checkout - but receive an error message: "error: Your local changes to the following files would be overwritten by checkout: NewFile.txt Please, commit your changes or stash them before you can switch branches. Aborting". This indicates that Git is preventing the switch because there are uncommitted changes.

To quickly resolve this situation, we can stash the modifications, putting them into a sort of box, ready to be unboxed at a later time.

Stashing is as simple as typing the `git stash` command. A default description will be added to your stash, and then modifications will be reverted to get back in a clean state:

```
MINGW32:/C/Repos/MyFirstRepo
Nando@LIAN /C/Repos/MyFirstRepo (master)
$ echo "New line added" >> NewFile.txt

Nando@LIAN /C/Repos/MyFirstRepo (master)
$ git checkout -
error: Your local changes to the following files would be overwritten by checkout:
  NewFile.txt
Please, commit your changes or stash them before you can switch branches.
Aborting

Nando@LIAN /C/Repos/MyFirstRepo (master)
$ git stash
Saved working directory and index state WIP on master: 32b25a3 Merge branch 'NewWork'
HEAD is now at 32b25a3 Merge branch 'NewWork'

Nando@LIAN /C/Repos/MyFirstRepo (master)
$ git status
On branch master
nothing to commit, working directory clean

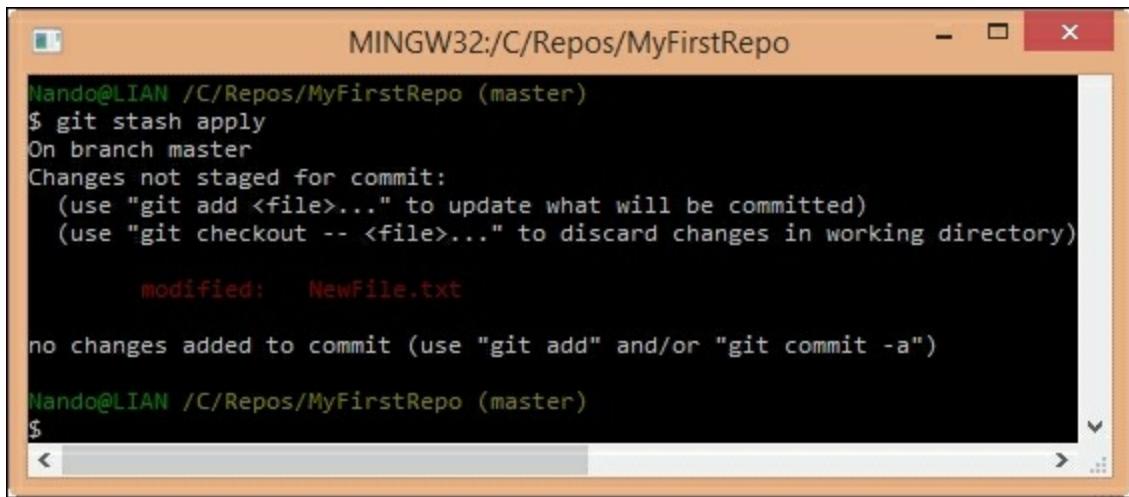
Nando@LIAN /C/Repos/MyFirstRepo (master)
$
```

To list actual stashes, you can use the `list` subcommand:

```
MINGW32:/C/Repos/MyFirstRepo
Nando@LIAN /C/Repos/MyFirstRepo (master)
$ git stash list
stash@{0}: WIP on master: 32b25a3 Merge branch 'NewWork'

Nando@LIAN /C/Repos/MyFirstRepo (master)
$
```

Once you have done the other work, you can go back to the previous branch and apply the stash to get back to the previous "work in progress" situation:



The screenshot shows a terminal window titled "MINGW32:/C/Repos/MyFirstRepo". The command \$ git stash apply was run, resulting in the following output:

```
Nando@LIAN /C/Repos/MyFirstRepo (master)
$ git stash apply
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   NewFile.txt

no changes added to commit (use "git add" and/or "git commit -a")
Nando@LIAN /C/Repos/MyFirstRepo (master)
$
```

What we have seen is the most common scenario and most used approach, but stashing is a powerful Git tool. You can have multiple stashes, apply a stash to a different branch, or reverse apply a stash. You can even create a branch starting from a stash. You can learn more about this on your own.

Summary

In this chapter, you learned the core concepts of Git: how it handles files and folders, how to include or exclude files in commits that we do, and how commits compose a Git repository.

Next, we explored the most used and powerful feature of Git, its ability to manage multiple parallel branches. For a developer, this is the feature that saves you time and headache while working on different parts of your project. This feature lets you and your colleagues collaborate without conflicts.

At the end, we touched on Git's stashing ability, where you can freeze your current work without having to commit an unfinished change. This helps you develop good programmer habits, such as the one that tells you to not commit an unfinished change.

In the next chapter, we will complete our journey of Git fundamentals, exploring ways and techniques to collaborate with other people.

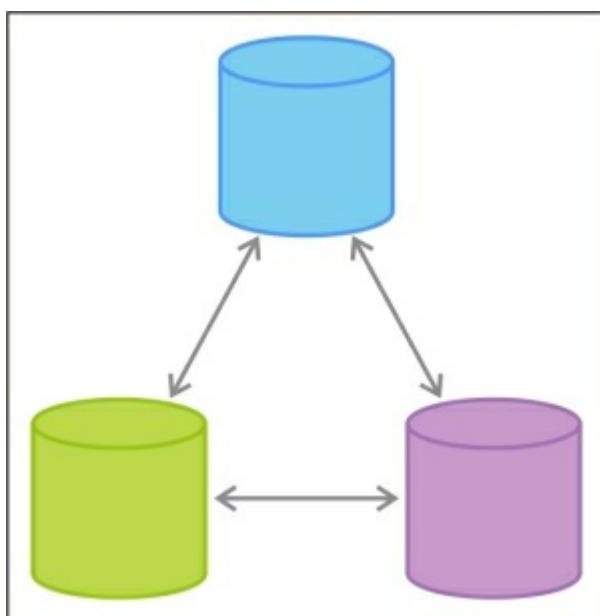
Chapter 3. Git Fundamentals – Working Remotely

In this chapter, we will finally start to work in a distributed manner, using remote servers as a contact point for different developers. As we said earlier, Git is a distributed version-control system. This chapter is for the distributed part.

Working with remotes

You know that Git is a tool for versioning files. However, it has been built with collaboration in mind. In 2005, Linus Torvalds had the need for a light and efficient tool to handle tons of patches proposed to the Linux kernel from a multitude of contributors. He wanted a tool that would allow him and hundreds of other people to work on it without going crazy. The pragmatism that guided its development gave us a very robust layer to share data among computers, without the need of a central server.

A Git remote is another computer that has the same repository you have on your computer. Every computer that hosts the same repository on a shared network can be the remote of other computers:



So, a remote Git repository is nothing other than a remote copy of the same Git repository we created locally. If you have access to that host via common protocols such as SSH, HTTPS or the custom `git://` protocol, you can keep your modification with it in sync.

The great difference between Git and other **distributed version control systems (DVCS)** to classical centralized versioning systems (VCS) such as Subversion is that there's no central server where you can give custody of your repository. However, you can have many remote servers. This allows you to be fault tolerant and have multiple working copies where you can get or pull modifications, giving you incredible flexibility.

To start working with a remote, we have to get one. Today, it is not difficult to get a remote. The world has plenty of free online services that offer room for Git repositories. One of the most commonly used repository is **GitHub**. Before starting, note that GitHub offers free space for open source repositories, so everyone in the world can access their code. Be careful, and don't store sensitive information such as passwords and so on in your repository; they will be publicly visible.

Setting up a new GitHub account

GitHub offers unlimited free public repositories, so we can make use of them without investing a penny. In GitHub, you have to pay only if you need private repositories, for example, to store the closed source code on which you base your business.

Creating a new account is simple. Just perform the following steps:

1. Go to <https://github.com>.
2. Sign up using your e-mail.

GitHub Search GitHub Explore Features Enterprise Blog Sign in

Build software better, together.

Powerful collaboration, code review, and code management for open source and private projects. Need private repositories? Upgraded plans start at \$7/mo.

Pick a username
Your email
Create a password
Use at least one lowercase letter, one numeral, and seven characters.
Sign up for GitHub

By clicking "Sign up for GitHub", you agree to our [terms of service](#) and [privacy policy](#). We will send you account related emails occasionally.

When done, we are ready to create a brand new repository where we can push our work:

Search GitHub Explore Gist Blog Help fsantacroce + - ⌂ ⚙ ⌂



Ferdinando Santacroce
fsantacroce

Italy
ferdinando.santacroce@gmail.com
<http://jesuswasrasta.com>
Joined on Nov 23, 2014

0 Followers 0 Starred 0 Following

Contributions Repositories Public activity [Edit profile](#)

Contributions

Dec	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov
M											
W											
F											

Summary of Pull Requests, issues opened, and commits. [Learn more.](#) Less More

This is your **contribution graph**. When you make a commit to a repository, you'll get a ■ for that day. Make more contributions and you'll get a darker green square. Over time, your chart might start looking [something like this](#).

We have a quick guide that will show you how to create your first repository. You'll also make a commit and **earn your first green square!**

[Read the Hello World guide](#)

Go to the **Repositories** tab, click on the green **New** button, and choose a name for your repository:

A screenshot of a GitHub profile page for the user 'fsantacroce'. The top navigation bar includes links for Explore, Gist, Blog, Help, and a user icon. Below the navigation is a header with 'Repositories' and 'Public activity' buttons, and an 'Edit profile' button. A search bar is followed by filters for All, Public, Private, Sources, Forks, and Mirrors, with a 'New' button. The main content area displays the message: 'fsantacroce doesn't have any public repositories yet.'

For the purpose of learning, I will create a simple repository for my personal recipes. These recipes are written using the Markdown markup language (<http://daringfireball.net/projects/markdown/>).

The screenshot shows the 'Create repository' form on GitHub. It includes fields for 'Owner' (set to 'fsantacroce'), 'Repository name' ('Cookbook'), and a checked 'Initialize this repository with a README' checkbox. Other visible options include 'Add .gitignore' (None), 'Add a license' (None), and a large green 'Create repository' button at the bottom.

Owner: fsantacroce / Repository name: Cookbook

Great repository names are short and memorable. Need inspiration? How about [yolo-ironman](#).

Description (optional): This repository contains recipes I like to share with my friends

Public
Anyone can see this repository. You choose who can commit.

Private
You choose who can see and commit to this repository.

Initialize this repository with a README
This will allow you to `git clone` the repository immediately. Skip this step if you have already run `git init` locally.

Add .gitignore: **None** | Add a license: **None** ⓘ

Create repository

Then, you can write a description for your repository. This is useful to allow people who come to visit your profile to better understand what your project is intended for. We create our repository as public because private repositories have a cost, as we said earlier. Then, we initialize it with a `README` file. Choosing this GitHub makes a first commit for us, initializing the repository that is now ready to use.

Cloning a repository

Now, we have a remote repository, so it's time to learn how to hook it locally. For this, Git provides the `git clone` command.

Using this command is quite simple. All we need to know is the URL of the repository to clone. The URL is provided by GitHub in the bottom-right corner of the repository home page, as shown in the following screenshot:



To copy the URL, you can simply click on the clipboard button on the right-hand side of the textbox.

So, let's try to follow these steps together:

1. Go to the local root folder, `C:\Repos`, for the repositories.
2. Open a Bash shell within it.
3. Type `git clone https://github.com/fsantacroce/Cookbook.git`.

Obviously, the URL of your repository will be different. As you can see

in this screenshot, GitHub URLs are composed by

`https://github.com/<Username>/<RepositoryName>.git`:

```
MINGW32:/C/Repos/Cookbook
Welcome to Git (version 1.9.5-preview20141217)

Run 'git help git' to display the help index.
Run 'git help <command>' to display help for specific commands.

Nando@LIAN /C/Repos
$ git clone https://github.com/fsantacroce/Cookbook.git
Cloning into 'Cookbook'...
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
Checking connectivity... done.

Nando@LIAN /C/Repos
$ ls -l
total 6
drwxr-xr-x    4 Nando    Administ      0 Dec 21 17:16 Cookbook
drwxr-xr-x    9 Nando    Administ  12288 Nov 30 16:02 MyFirstRepo

Nando@LIAN /C/Repos
$ cd Cookbook/
Nando@LIAN /C/Repos/Cookbook (master)
$
```

At this point, Git created a new `Cookbook` folder that contains the downloaded copy of our repository. Inside, we will find a `README.md` file, a classical one for a GitHub repository. In this file, you can describe your repository using the common Markdown markup language to users who will chance upon it.

Uploading modifications to remotes

So, let's try to edit the `README.md` file and upload modifications to GitHub:

1. Edit the `README.md` file using your preferred editor. You can add, for example, a new sentence.
2. Add it to the index and then commit.
3. Put your commit on the remote repository using the `git push` command.

```
MINGW32:/C/Repos/Cookbook
Nando@LIAN /C/Repos/Cookbook (master)
$ vim README.md

Nando@LIAN /C/Repos/Cookbook (master)
$ git add README.md

Nando@LIAN /C/Repos/Cookbook (master)
$ git commit -m "Adds a sentence to readme"
[master e1e7236] Adds a sentence to readme
 1 file changed, 2 insertions(+), 1 deletion(-)

Nando@LIAN /C/Repos/Cookbook (master)
$ git push
warning: push.default is unset; its implicit value is changing in
Git 2.0 from 'matching' to 'simple'. To squelch this message
and maintain the current behavior after the default changes, use:

  git config --global push.default matching

To squelch this message and adopt the new behavior now, use:

  git config --global push.default simple

When push.default is set to 'matching', git will push local branches
to the remote branches that already exist with the same name.

In Git 2.0, Git will default to the more conservative 'simple'
behavior, which only pushes the current branch to the corresponding
remote branch that 'git pull' uses to update the current branch.

See 'git help config' and search for 'push.default' for further information.
(the 'simple' mode was introduced in Git 1.7.11. Use the similar mode
'current' instead of 'simple' if you sometimes use older versions of Git)

Username for 'https://github.com': fsantacroce
Password for 'https://fsantacroce@github.com':
Counting objects: 5, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 379 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/fsantacroce/Cookbook.git
  e6d7f59..e1e7236  master -> master

Nando@LIAN /C/Repos/Cookbook (master)
$
```

The `git push` command allows you to upload local work to a configured remote location, in this case, a remote GitHub repository. There are a few things we have to know about pushing. We can begin to understand the message Git gave us just after we run the `git push` command.

What do I send to the remote when I push?

When you give the `git push` command without specifying anything else,

Git sends to the remote all the new commits you did locally in your actual branch. For new commits, we will send only the local commits that have not been uploaded yet.

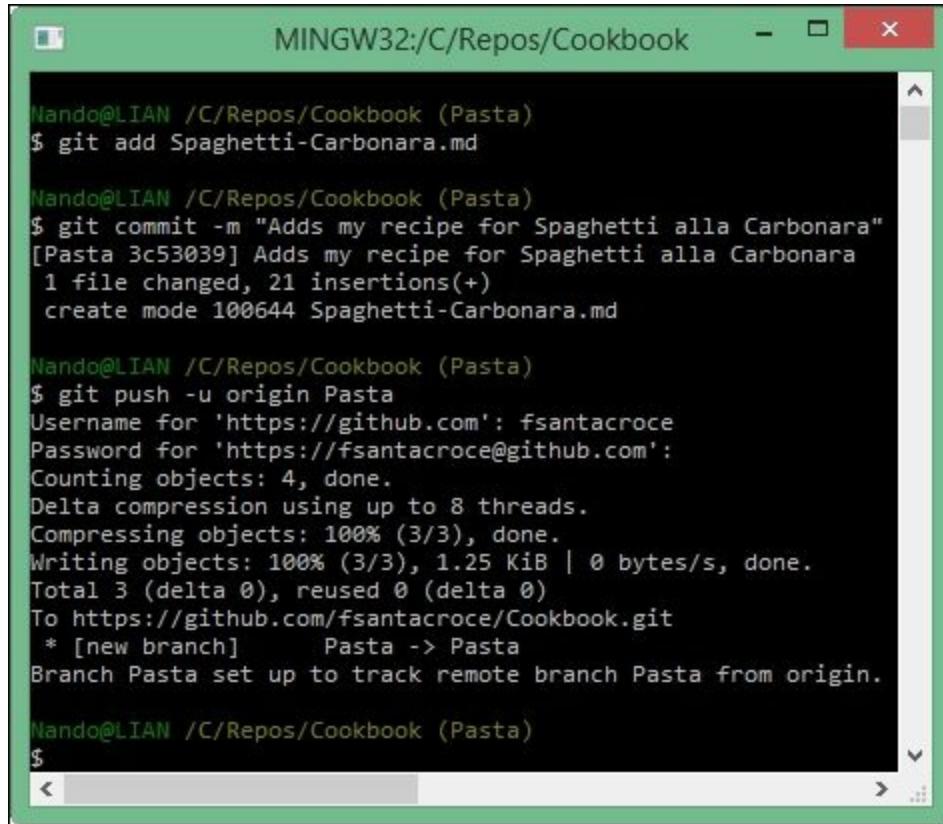
The messages that appeared earlier tell us that something is about to change in the default behavior of the `git push` command. Before the incoming Git 2.0 release, when you fire the `push` command without specifying anything else, the default behavior is to push all the new commits in all the corresponding local-remote branches.

In this brand new `Cookbook` repository, we only have the `master` branch. However, next, you will learn how to deal with the remote having many branches.

Pushing a new branch to the remote

Obviously, we can create and push a new branch to the remote to make our work public and visible to other collaborators. For instance, I will create a new branch for my first recipe; then, I will push this branch to the remote GitHub server. Follow these simple steps:

1. Create a new branch, for instance, `Pasta`: `git checkout -b Pasta`.
2. Add to it a new file, for example, `Spaghetti-Carbonara.md`, and commit it.
3. Push the branch to the remote using `git push -u origin Pasta`.



The screenshot shows a terminal window titled "MINGW32:/C/Repos/Cookbook". The session logs the following commands and their output:

```
Nando@LIAN /C/Repos/Cookbook (Pasta)
$ git add Spaghetti-Carbonara.md

Nando@LIAN /C/Repos/Cookbook (Pasta)
$ git commit -m "Adds my recipe for Spaghetti alla Carbonara"
[Pasta 3c53039] Adds my recipe for Spaghetti alla Carbonara
 1 file changed, 21 insertions(+)
 create mode 100644 Spaghetti-Carbonara.md

Nando@LIAN /C/Repos/Cookbook (Pasta)
$ git push -u origin Pasta
Username for 'https://github.com': fsantacroce
Password for 'https://fsantacroce@github.com':
Counting objects: 4, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 1.25 KiB | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/fsantacroce/Cookbook.git
 * [new branch]      Pasta -> Pasta
Branch Pasta set up to track remote branch Pasta from origin.

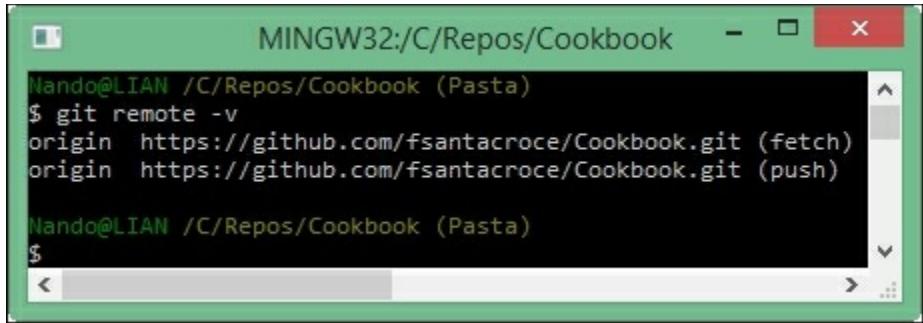
Nando@LIAN /C/Repos/Cookbook (Pasta)
$
```

Before continuing, we have to examine in depth some things that happened after we used the `git push` command.

The origin

With the `git push -u origin Pasta` command, we told Git to upload our `Pasta` branch (and the commits within it) to the origin. The origin is the default remote name of a repository, like `master` is the default branch name. When you clone a repository from a remote, that remote becomes your `origin` alias. When you tell Git to push or pull something, you often have to tell it about the remote you want to use. Using the `origin` alias, you tell Git that you want to use your default remote.

If you want to see remotes actually configured in your repository, you could type a simple `git remote` command, followed by `-v` (`--verbose`) to get some more details:



```
Nando@LIAN /C/Repos/Cookbook (Pasta)
$ git remote -v
origin  https://github.com/fsantacroce/Cookbook.git (fetch)
origin  https://github.com/fsantacroce/Cookbook.git (push)

Nando@LIAN /C/Repos/Cookbook (Pasta)
$
```

In the details, you will see the full URL of the remote.

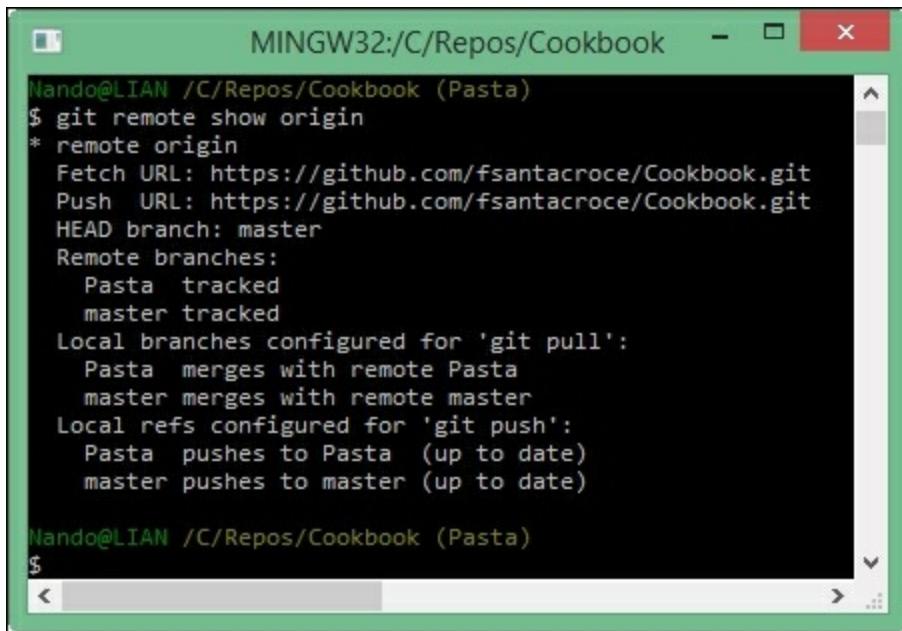
You can add, update, and delete remotes using the `git remote` command. We will make use of it when we need to. For now, just remember that there is a command for that.

Tracking branches

Using the `-u` option, we told Git to track the remote branch. Tracking a remote branch is the way to tie your local branch with the remote one. Note that this behavior is not automatic; you have to set it if you want it (Git does nothing until you ask for it!). When a local branch tracks a remote branch, you actually have a local and remote branch that can be kept easily in sync. This is very useful when you have to collaborate with some remote coworkers at the same branch, allowing all of them to keep their work in sync with other people's changes.

Furthermore, consider that when you use the `git fetch` command, you will get changes from all tracked branches. While you use the `git push` command, the default behavior is to push to the corresponding remote branch only.

To better understand the way our repository is now configured, try to type `git remote show origin`:



A screenshot of a terminal window titled "MINGW32:/C/Repos/Cookbook". The window displays the output of the command \$ git remote show origin. The output shows that the local master branch tracks the remote Pasta branch, and both branches are up-to-date. Local branches Pasta and master merge with their respective remote branches.

```
Nando@LIAN /C/Repos/Cookbook (Pasta)
$ git remote show origin
* remote origin
  Fetch URL: https://github.com/fsantacroce/Cookbook.git
  Push  URL: https://github.com/fsantacroce/Cookbook.git
  HEAD branch: master
  Remote branches:
    Pasta tracked
    master tracked
  Local branches configured for 'git pull':
    Pasta merges with remote Pasta
    master merges with remote master
  Local refs configured for 'git push':
    Pasta pushes to Pasta (up to date)
    master pushes to master (up to date)

Nando@LIAN /C/Repos/Cookbook (Pasta)
$
```

As you can see, both the `Pasta` and `master` branches are tracked.

You can also see that your local branches are configured to push and pull to remote branches with the same name. However, remember that it is not mandatory to have local and remote branches with the same name. The local branch `foo` can track the remote branch `bar` and vice versa; there are no restrictions.

Downloading remote changes

The first thing you have to learn when working with remote branches is to check whether there are modifications to retrieve.

Checking for modifications and downloading them

The `git fetch` command is very helpful. To see it at work, create a new file directly on GitHub using the following steps so that we can work with it:

1. Go to your GitHub dashboard and choose the `Pasta` branch.
2. Click the plus icon on the right-hand side of the repository name to add a new file.



3. Add a new empty file, for example, `Bucatini-Amatriciana.md`, and commit it directly from GitHub:

The screenshot shows the GitHub interface for creating a new file. At the top, there's a header with 'Cookbook / Bucatini-Amatriciana.md' and a 'cancel' link. Below that is a code editor window with two lines of code: '1 # Bucatini Amatriciana' and '2 |'. To the right of the editor are settings for 'Spaces' (set to 2), 'No wrap', and a copy icon. Below the editor is a modal titled 'Commit new file' with a placeholder 'Create Bucatini-Amatriciana.md' and a larger text area for an optional extended description. At the bottom of the modal are 'Commit new file' and 'Cancel' buttons.

Now, we can make use of `git fetch`:

```
MINGW32:/C/Repos/Cookbook
Nando@LIAN /C/Repos/Cookbook (Pasta)
$ git fetch
From https://github.com/fsantacroce/Cookbook
 3c53039..83c6321 Pasta -> origin/Pasta
Nando@LIAN /C/Repos/Cookbook (Pasta)
$
```

The `git fetch` command downloads differences from the remote, but does not apply them. With the help of `git status`, we will see more details:



```
MINGW32:/C/Repos/Cookbook
Nando@LIAN /C/Repos/Cookbook (Pasta)
$ git status
On branch Pasta
Your branch is behind 'origin/Pasta' by 1 commit, and can be fast-forwarded.
(use "git pull" to update your local branch)

nothing to commit, working directory clean

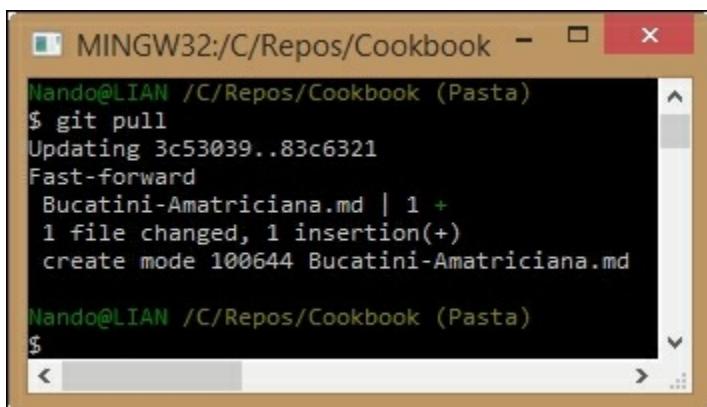
Nando@LIAN /C/Repos/Cookbook (Pasta)
$
```

As you can see, Git tells us there is one downloaded commit we have to apply to keep our branch in sync with the remote counterpart.

Applying downloaded changes

If **push** is the verb used to define the upload part of the work, **pull** is the verb used to describe the action of downloading and applying remote changes. When you pull something from a remote, Git will retrieve all the remote commits made after your last pull from the branch you specify and merge them. Of course, the local destination branch is the branch you are in now (if you don't explicitly specify another one). So, finally, the pull command is the `git fetch` command plus the `git merge` command (in the future, you can skip `git fetch` and use only `git pull` to merge remote commits).

Now, use the `git pull` command to merge the brand new `Bucatini-Amatriciana.md` file committed directly from GitHub:

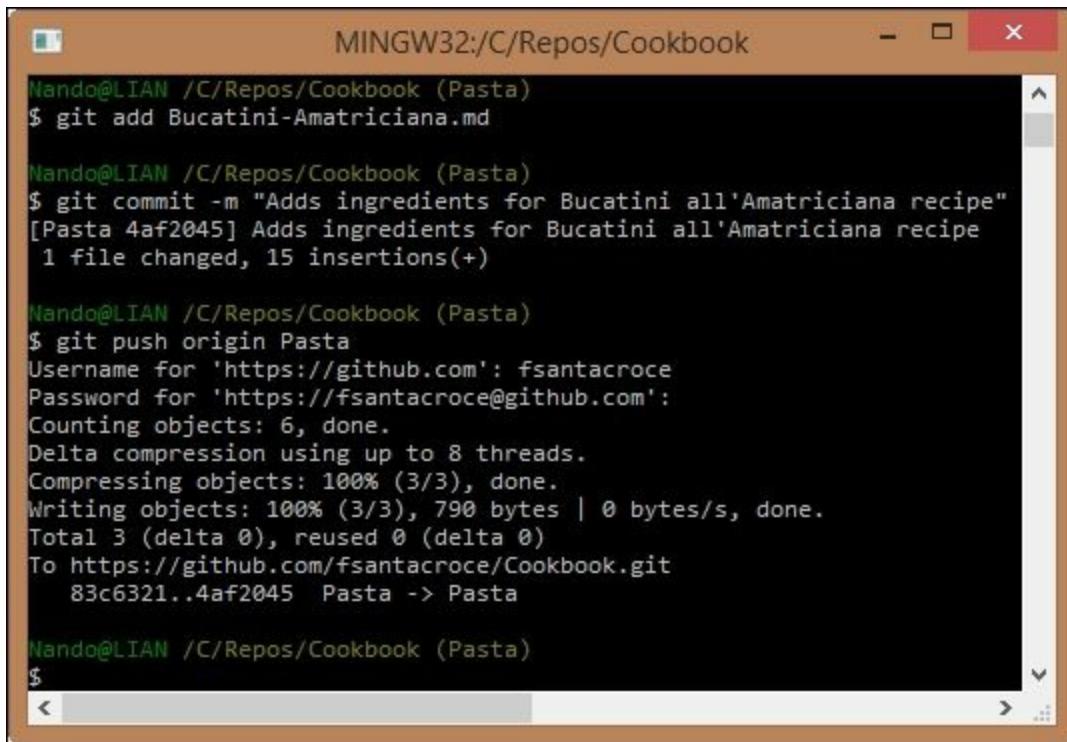


```
MINGW32:/C/Repos/Cookbook
Nando@LIAN /C/Repos/Cookbook (Pasta)
$ git pull
Updating 3c53039..83c6321
Fast-forward
Bucatini-Amatriciana.md | 1 +
1 file changed, 1 insertion(+)
create mode 100644 Bucatini-Amatriciana.md

Nando@LIAN /C/Repos/Cookbook (Pasta)
$
```

Well done! Now, our `Pasta` branch is in sync with the remote one. We have the `Bucatini-Amatriciana.md` file in our local branch, ready to be filled.

Just to end this simple example, make some modifications to the new downloaded file and push the commit to GitHub:



```
MINGW32:/C/Repos/Cookbook
Nando@LIAN /C/Repos/Cookbook (Pasta)
$ git add Bucatini-Amatriciana.md

Nando@LIAN /C/Repos/Cookbook (Pasta)
$ git commit -m "Adds ingredients for Bucatini all'Amatriciana recipe"
[Pasta 4af2045] Adds ingredients for Bucatini all'Amatriciana recipe
 1 file changed, 15 insertions(+)

Nando@LIAN /C/Repos/Cookbook (Pasta)
$ git push origin Pasta
Username for 'https://github.com': fsantacroce
Password for 'https://fsantacroce@github.com':
Counting objects: 6, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 790 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/fsantacroce/Cookbook.git
 83c6321..4af2045  Pasta -> Pasta

Nando@LIAN /C/Repos/Cookbook (Pasta)
$
```

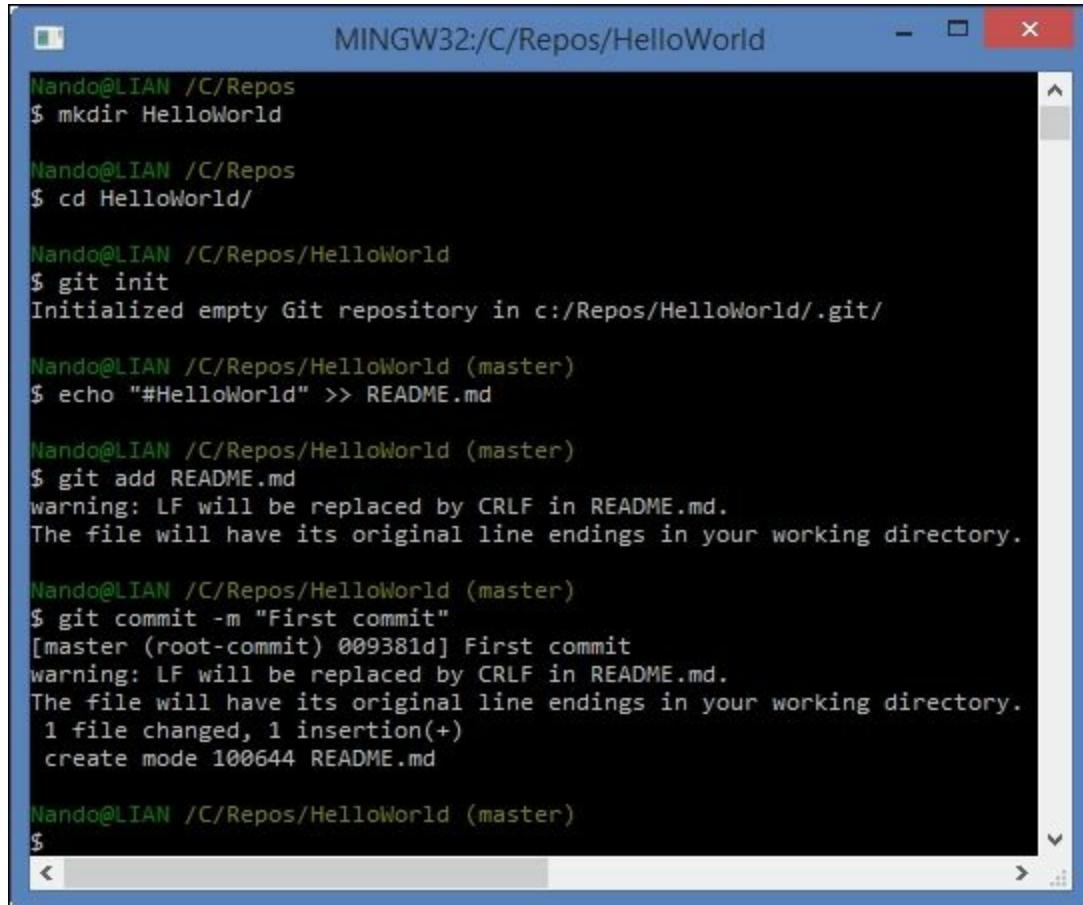
This time we have specified the branch where we want to push changes, so Git avoids reminding us the default `git push` behavior is going to change in the next release.

Going backward: publish a local repository to GitHub

Often, you will find yourself needing to put your local repository in a shared place where someone else can look at your work. In this section, we will learn how to achieve this.

Create a new local repository to publish by following these simple steps:

1. Go to our C:\Repos folder.
2. Create a new HelloWorld folder.
3. Init a new repository, as we did in [Chapter 1, Getting Started with Git](#).
4. Add a new README.md file and commit it.



The screenshot shows a terminal window titled "MINGW32:/C/Repos>HelloWorld". The session logs the following commands and their output:

```
Nando@LIAN /C/Repos
$ mkdir HelloWorld

Nando@LIAN /C/Repos
$ cd HelloWorld/

Nando@LIAN /C/Repos>HelloWorld
$ git init
Initialized empty Git repository in c:/Repos>HelloWorld/.git/

Nando@LIAN /C/Repos>HelloWorld (master)
$ echo "#HelloWorld" >> README.md

Nando@LIAN /C/Repos>HelloWorld (master)
$ git add README.md
warning: LF will be replaced by CRLF in README.md.
The file will have its original line endings in your working directory.

Nando@LIAN /C/Repos>HelloWorld (master)
$ git commit -m "First commit"
[master (root-commit) 009381d] First commit
warning: LF will be replaced by CRLF in README.md.
The file will have its original line endings in your working directory.
 1 file changed, 1 insertion(+)
 create mode 100644 README.md

Nando@LIAN /C/Repos>HelloWorld (master)
$
```

Now, create the GitHub repository as we did previously. This time, leave it empty. Don't initialize it with a README.md file; we already have one in our local repository:

Owner  fsantacroce / Repository name ✓

Great repository names are short and memorable. Need inspiration? How about `bugfree-octo-ninja`.

Description (optional)
A simple repository for tests

 **Public**
Anyone can see this repository. You choose who can commit.

 **Private**
You choose who can see and commit to this repository.

Initialize this repository with a README
This will allow you to `git clone` the repository immediately. Skip this step if you have already run `git init` locally.

Add .gitignore: **None** | Add a license: **None** ⓘ

Create repository

At this point, we are ready to publish our local repository on GitHub or, more specifically, add a remote to it.

Adding a remote to a local repository

To publish our `HelloWorld` repository, we simply have to add its first remote. Adding a remote is quite simple: Use the command `git remote add origin <remote-repository-url>`.

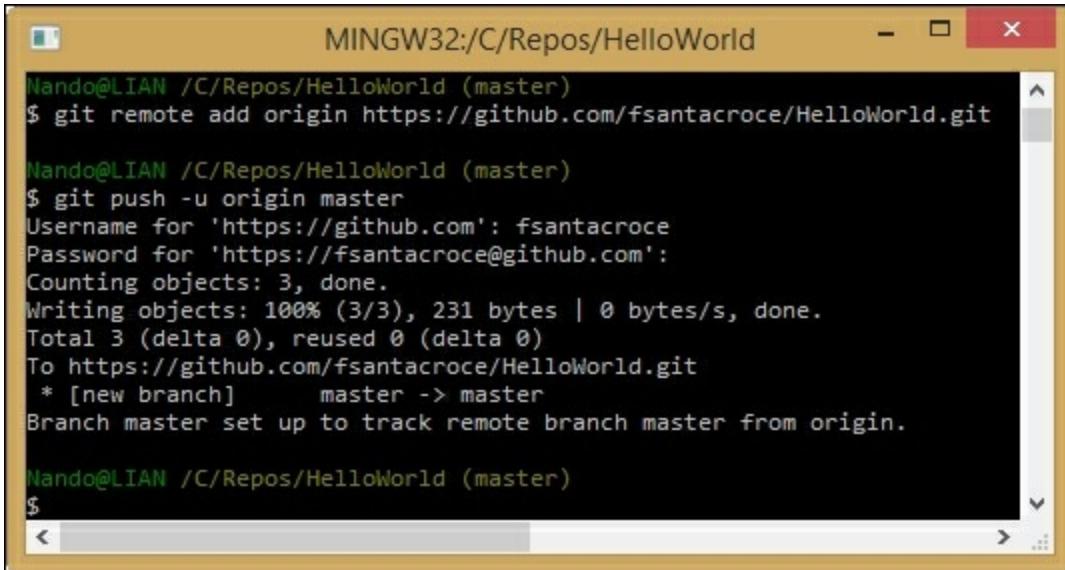
So, type `git remote add origin https://github.com/fsantacroce>HelloWorld.git` in the Bash shell.

Adding a remote, like adding or modifying other configuration parameters, simply consists of editing some text files in the `.git` folder. In the next chapter, we will take a look at some of these files.

Pushing a local branch to a remote repository

After adding a remote, push your local changes to the remote using `git`

```
push -u origin master:
```



```
Nando@LIAN /C/Repos/HelloWorld (master)
$ git remote add origin https://github.com/fsantacroce/HelloWorld.git

Nando@LIAN /C/Repos/HelloWorld (master)
$ git push -u origin master
Username for 'https://github.com': fsantacroce
Password for 'https://fsantacroce@github.com':
Counting objects: 3, done.
Writing objects: 100% (3/3), 231 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/fsantacroce/HelloWorld.git
 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.

Nando@LIAN /C/Repos/HelloWorld (master)
$
```

That's all!

Social coding – collaborate using GitHub

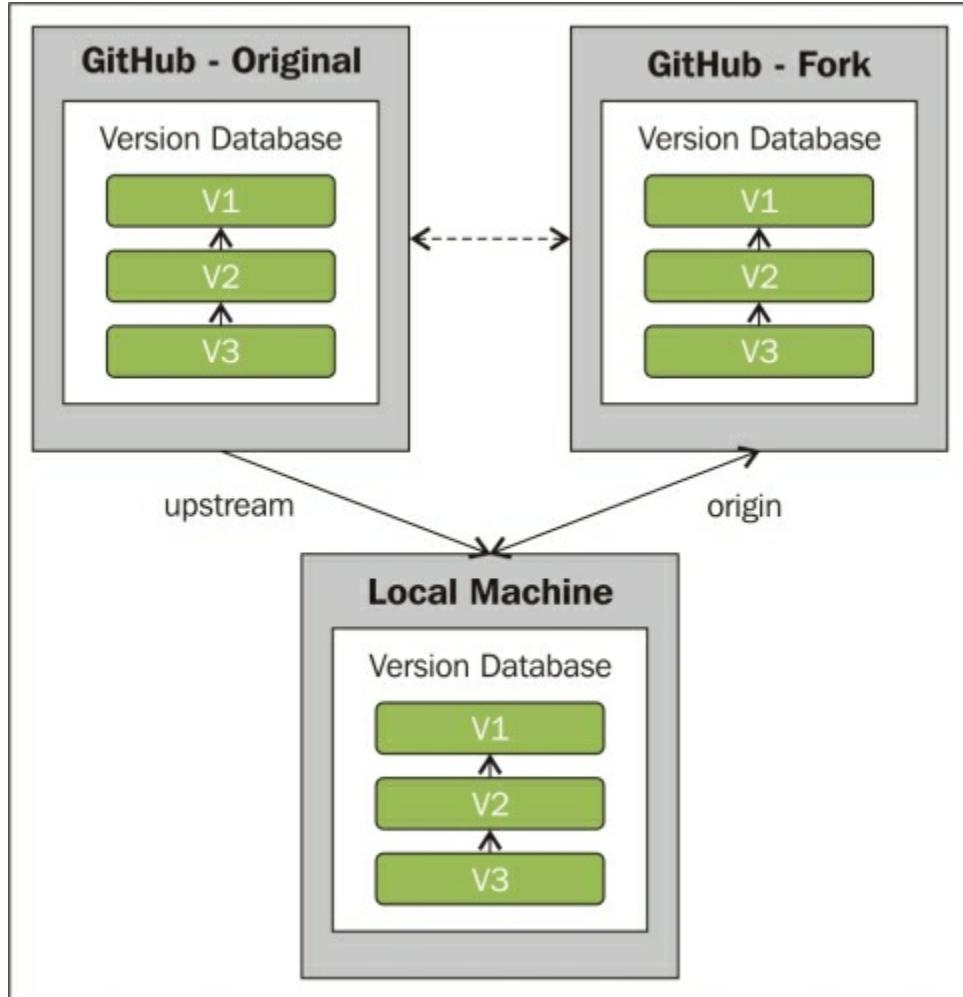
GitHub's trademark is the so-called **social coding**. Right from the beginning, GitHub made it easy to share code, track other people's work, and collaborate using two basic concepts: forks and pull requests. In this section, I will illustrate these concepts in brief.

Forking a repository

Forking is a common concept for a developer. If you have already used a GNU-Linux-based distribution, you would know that there are some forefathers, such as Debian, and some derived distro, such as Ubuntu. They are commonly called the **forks** of the original distribution.

In GitHub, things are similar. At some point, you will find an interesting open-source project that you want to modify slightly to perfectly fit your needs. At the same time, you want to benefit from bug fixes and new features from the original project, all the time keeping in touch with your work. The right thing to do in this situation is to fork the project.

However, first, remember that fork is not a Git feature, but a GitHub invention. When you fork on GitHub, you get a server-side clone of the repository on your GitHub account. If you clone your forked repository locally in the remote list, you would find the `origin` alias that points to your account repository. The original repository will generally assume the `upstream` alias (but this is not automatic, you have to add it manually):



For your convenience, the right command to add the `upstream` remote is
\$ git remote add upstream https://github.com/<original-owner>/<original-repository>.git.

Now, to better understand this feature, go to your GitHub account and try to fork a common GitHub repository called `Spoon-Knife`, made by the `octocat` GitHub mascot user. Perform the following steps:

1. Log in to your GitHub account.

2. Look for spoon-knife using the search textbox located in the top-left corner of the page:



3. Click on the first result, octocat/Spoon-Knife repository.
4. Fork the repository using the Fork button at the right of the page:



5. After a funny photocopy animation, you will get a brand new Spoon-Knife repository on your GitHub account:



Now, you can clone this repository locally, as we did earlier:

```
Select MINGW32:/C/Repos/Spoon-Knife
Nando@LIAN /C/Repos
$ git clone https://github.com/fsantacroce/Spoon-Knife.git
Cloning into 'Spoon-Knife'...
remote: Counting objects: 16, done.
remote: Total 16 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (16/16), done.
Checking connectivity... done.

Nando@LIAN /C/Repos
$ cd Spoon-Knife

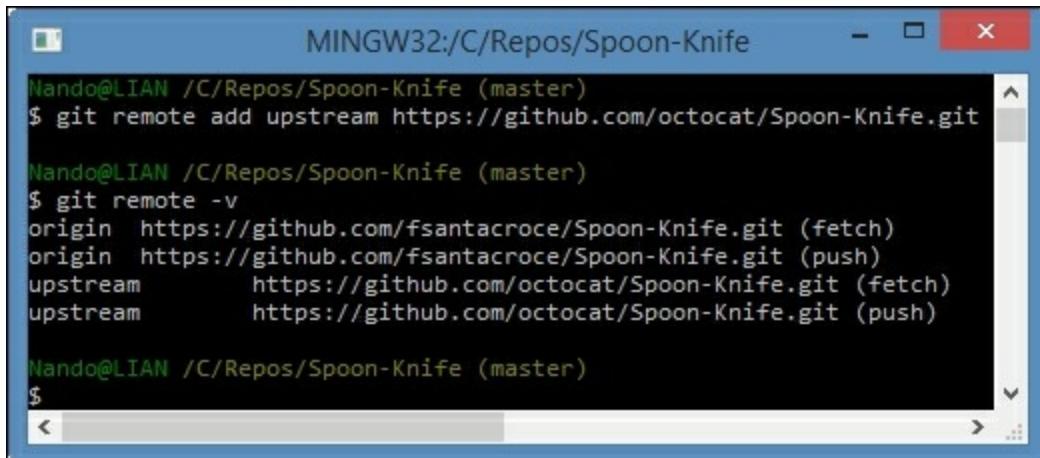
Nando@LIAN /C/Repos/Spoon-Knife (master)
$ git remote -v
origin  https://github.com/fsantacroce/Spoon-Knife.git (fetch)
origin  https://github.com/fsantacroce/Spoon-Knife.git (push)

Nando@LIAN /C/Repos/Spoon-Knife (master)
$
```

A screenshot of a terminal window titled "Select MINGW32:/C/Repos/Spoon-Knife". The window shows a command-line session where the user is cloning the "Spoon-Knife" repository from GitHub. The output of the "git clone" command is displayed, showing the progress of cloning 16 objects and checking connectivity. The user then changes directory into the cloned repository and checks the remote branches. The terminal window has a blue header bar and a dark background.

As you can see, the `upstream` remote is not present. This is a convention, not a thing that belongs to Git itself. To add this remote, type:

```
git remote add upstream https://github.com/octocat/Spoon-Knife.git
```



The screenshot shows a terminal window with the title 'MINGW32:C/Repos/Spoon-Knife'. The command \$ git remote add upstream https://github.com/octocat/Spoon-Knife.git was run, and the output shows the new remote 'upstream' added. The terminal also lists the existing remote 'origin'.

```
Nando@LIAN /C/Repos/Spoon-Knife (master)
$ git remote add upstream https://github.com/octocat/Spoon-Knife.git

Nando@LIAN /C/Repos/Spoon-Knife (master)
$ git remote -v
origin  https://github.com/fsantacroce/Spoon-Knife.git (fetch)
origin  https://github.com/fsantacroce/Spoon-Knife.git (push)
upstream     https://github.com/octocat/Spoon-Knife.git (fetch)
upstream     https://github.com/octocat/Spoon-Knife.git (push)

Nando@LIAN /C/Repos/Spoon-Knife (master)
$
```

Now, you can keep your local repository in sync with the changes in your remote and the `origin` alias. You can also get changes ones coming from the `upstream` remote, the original repository you forked. At this point, you are probably wondering how to deal with two different remotes. Well, it is easy. Simply pull from `upstream` and merge those modifications in your local repository. Then, push them in your `origin` remote in addition to your changes. If someone else clones your repository, they would receive your work merged with the work done by someone else on the original repository.

Submitting pull requests

If you created a fork of a repository, this is because you are not a direct contributor of the original project or simply because you don't want to make a mess in other people's work before you are familiar with the code.

However, at a certain point, you realize that your work can be useful even for the original project. You realize that you can implement a

previous piece of code in a better way, you can add a missing feature, and so on.

So, you find yourself needing to allow the original author to know you did something interesting, to ask them if they want to take a look and, maybe, integrate your work. This is the moment when pull requests come in handy.

A pull request is a way to tell the original author, "Hey! I did something interesting using your original code. Do you want to take a look and integrate my work, if you find it good enough?" This is not only a technical way to achieve the purpose of integrating work, but it is even a powerful practice to promote **code reviews** (and then the so-called social coding) as recommended by the eXtreme Programming fellows (http://en.wikipedia.org/wiki/Extreme_programming).

One other reason to use a pull request is because you cannot push directly to the `upstream` remote if you are not a contributor of the original project. Pull requests are the only way. In small scenarios (such as a team of two or three developers that works in the same room), probably, the `fork` and `pull` model represents an overhead. So, it is more common to directly share the original repository with all the contributors, skipping the fork and pull ceremony.

Creating a pull request

To create a pull request, you have to go to your GitHub account and make it directly from your forked account. However, first, you have to know that pull requests can be made only from separated branches. At this point of the book, you are probably used to creating a new branch for a new feature or refactor purpose. So, this is nothing new.

To try, let's create a local `TeaSpoon` branch in our repository, commit a new file, and push it to our GitHub account:

```
MINGW32:/C/Repos/Spoon-Knife

Nando@LIAN /C/Repos/Spoon-Knife (master)
$ git branch TeaSpoon

Nando@LIAN /C/Repos/Spoon-Knife (master)
$ git checkout TeaSpoon
Switched to branch 'TeaSpoon'

Nando@LIAN /C/Repos/Spoon-Knife (TeaSpoon)
$ vim TeaSpoon.md

Nando@LIAN /C/Repos/Spoon-Knife (TeaSpoon)
$ git add TeaSpoon.md

Nando@LIAN /C/Repos/Spoon-Knife (TeaSpoon)
$ git commit -m "Adds a TeaSpoon to the cutlery"
[TeaSpoon 7ec6db1] Adds a TeaSpoon to the cutlery
 1 file changed, 2 insertions(+)
 create mode 100644 TeaSpoon.md

Nando@LIAN /C/Repos/Spoon-Knife (TeaSpoon)
$ git push -u origin TeaSpoon
Username for 'https://github.com': fsantacroce
Password for 'https://fsantacroce@github.com':
Counting objects: 6, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 423 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/fsantacroce/Spoon-Knife.git
 * [new branch]      TeaSpoon -> TeaSpoon
Branch TeaSpoon set up to track remote branch TeaSpoon from origin.

Nando@LIAN /C/Repos/Spoon-Knife (TeaSpoon)
$
```

If you take a look at your account, you will find a surprise: in your Spoon-Knife repository, there is now a new green button made on purpose to start a pull request:

The screenshot shows a GitHub repository page for 'fsantacroce / Spoon-Knife'. The repository was forked from 'octocat/Spoon-Knife'. The page includes a brief description: 'This repo is for demonstration purposes only.' Below the description are summary statistics: 3 commits, 3 branches, 0 releases, and 1 contributor. A green 'Compare & pull request' button is prominently displayed. Under the 'Your recently pushed branches:' section, the 'TeaSpoon' branch is listed with a timestamp of 'less than a minute ago'. At the bottom, there are navigation links for 'Issues', 'Branch: master', and 'Spoon-Knife / +'.

Clicking on this button makes GitHub open a new page where we can adorn our pull request to better support our work. We can let the original author know why we think our work can be useful even in the original project.

However, let me analyze this new page in brief.

In the top-left corner, you will find what branches GitHub is about to compare for you:



This means that you are about to compare your local `TeaSpoon` branch with the original `master` branch of the `octocat` user. At the bottom of the page, you can see all the different details (files added, removed, changed, and so on):



In the central part of the page, you can describe the work you did in your branch. After that, you have to click on the green **Create pull request** button to send your request to the original author, allowing him to access your work and analyze it. A green **Able to merge** sticker on the right-hand side informs you that these two branches can be automatically merged (there are no unresolved conflicts, which is always

good to see, considering your work):

The screenshot shows a GitHub pull request interface. At the top, there's a header with the title "Adds a TeaSpoon to the cutlery". Below the header, there are two tabs: "Write" (selected) and "Preview". To the right of the tabs are "Markdown supported" and "Edit in fullscreen" buttons. On the far right, there's a green icon of two interlocking branches and the text "Able to merge". Below the header, the main content area contains a message from "fsantacroce": "Hello! I extended the repository including a teaspoon; hope you find it useful :). Bye, Nando". There's also a note at the bottom of the message area: "Attach images by dragging & dropping, selecting them, or pasting from the clipboard." On the right side of the message area, there's a green "Create pull request" button.

Now, the last step: click on the **Create pull request** button and cross your fingers.

The screenshot shows the details of a GitHub pull request titled "Adds a TeaSpoon to the cutlery #4254". The pull request is open and merging "fsantacroce" into "octocat:master" from "fsantacroce:TeaSpoon". The conversation tab shows one comment from "fsantacroce" just now, which includes the message: "Hello! I extended the repository including a teaspoon; hope you find it useful :). Bye, Nando". Below the comment, there's a commit message: "Adds a TeaSpoon to the cutlery" with the SHA "7ec6db1". A note below the commit says "Add more commits by pushing to the TeaSpoon branch on fsantacroce/Spoon-Knife.". At the bottom, there's a green icon of two interlocking branches and the text "This pull request can be automatically merged by project collaborators. Only those with write access to this repository can merge pull requests.", with a "Merge" button to the right.

At this point a new conversation begins. You and the project collaborators can start to discuss your work. During this period, you and other collaborators can change the code to better fit common needs, until an original repository collaborator decides to accept your request or discard it, thus closing the pull request.

Summary

In this chapter, we finally got in touch with the Git ability to manage multiple remote copies of repositories. This gives you a wide range of possibilities to better organize your collaboration workflow inside your team.

In the next chapter, you will learn some advanced techniques using well-known and niche commands. This will make you a more secure and proficient Git user, allowing you to easily resolve some common issues that occur in a developer's life.

Chapter 4. Git Fundamentals – Niche Concepts, Configurations, and Commands

This chapter is a collection of short but useful tricks to make our Git experience more comfortable. In the first three chapters, we learnt all the concepts we need to take the first steps into versioning systems using the Git tool; now it's time to go a little bit more in depth to discover some other powerful weapons in the Git arsenal and see how to use them (without shooting yourself in your foot, preferably).

Dissecting the Git configuration

In the first part of this chapter, we will learn how to enhance our Git configuration to better fit our needs and speed up the daily work; now it's time to become familiar with the configuration internals.

Configuration architecture

The configuration options are stored in plain text files. The `git config` command is just a convenient tool to edit these files without the hassle of remembering where they are stored and opening them in a text editor.

Configuration levels

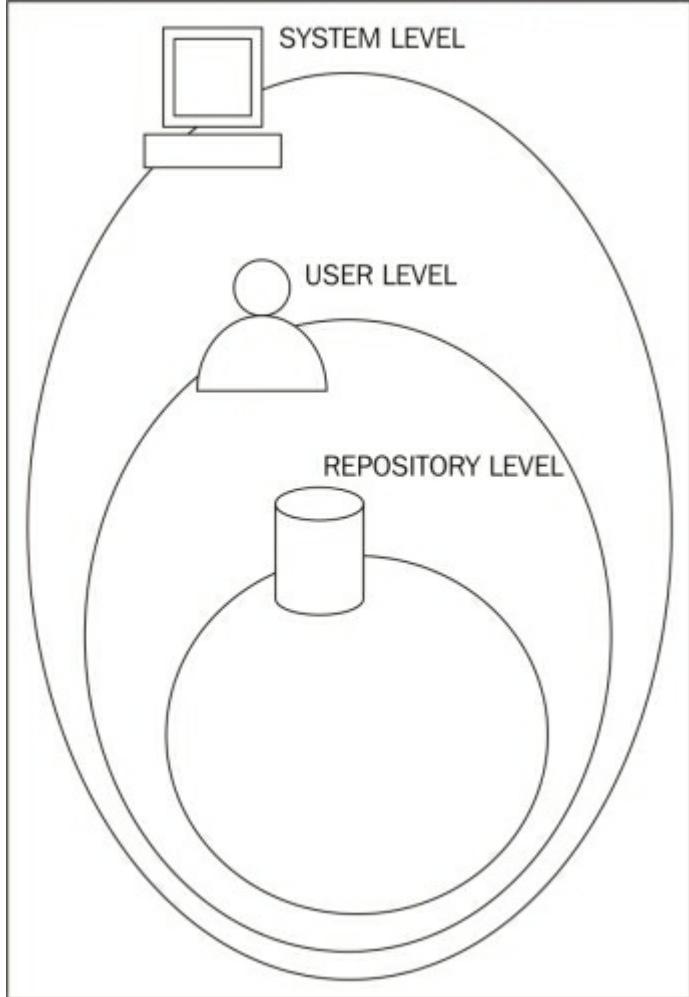
In Git we have three configuration levels which are:

- System
- User
- Repository

There are different configuration files for every different configuration level.

You can basically set every parameter at every level according to your

needs. If you set the same parameters at different levels, the lowest-level parameter hides the top level parameters; so, for example, if you set `user.name` at global level, it will hide the one eventually set up at system level; if you set it at repository level, it will hide the one specified at global level and the one eventually set up at system level.



System level

The system level contains **system-wide configurations**; if you edit the configuration at this level, every user and its repository will be affected.

This configuration is stored in the `gitconfig` file usually located in:

- Windows - `C:\Program Files (x86)\Git\etc\gitconfig`
- Linux - `/etc/gitconfig`
- Mac OS X - `/usr/local/git/etc/gitconfig`

To edit the parameters at this level, you have to use the `--system` option; please note that it requires administrative privileges (for example, root permission on Linux and Mac OS X). Anyway, as a rule of thumb, the edit configuration at system level is discouraged in favor of per user configuration modification.

Global level

The global level contains **user-wide configurations**; if you edit the configuration at this level, every user's repository will be affected.

This configuration is stored in the `.gitconfig` file usually located in:

- Windows - `C:\Users\<UserName>\.gitconfig`
- Linux - `~/.gitconfig`
- Mac OS X - `~/.gitconfig`

To edit the parameters at this level, you have to use the `--global` option.

Repository level

The repository level contains **repository only configurations**; if you edit the configuration at this level, only the repository in use will be affected.

This configuration is stored in the `config` file located in the `.git` repository subfolder:

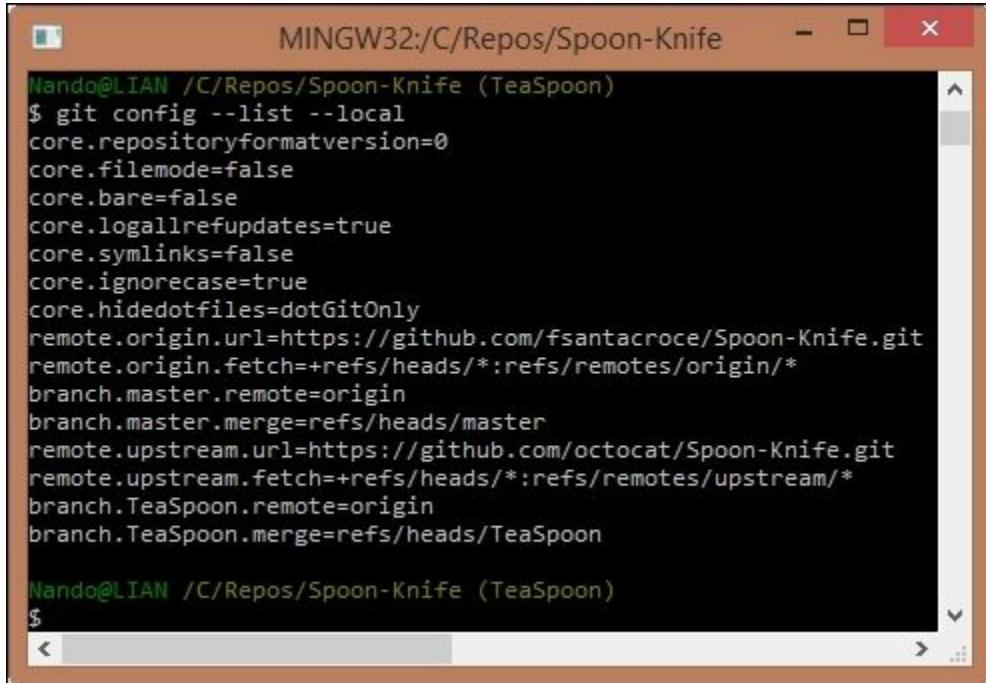
- Windows - `C:\<MyRepoFolder>\.git\config`
- Linux - `~/<MyRepoFolder>/.git/config`
- Mac OS X - `~/<MyRepoFolder>/.git/config`

To edit parameters at this level, you can use the `--local` option or simply avoid using any option as this is the default one.

Listing configurations

To get a list of all configurations currently in use, you can run the `git config --list` option; if you are inside a repository, it will show all the configurations from repository to system level. To filter the list, append -

-system, --global, or --local options to obtain only the desired level configurations, as shown in the following screenshot:



A screenshot of a terminal window titled "MINGW32:/C/Repos/Spoon-Knife (TeaSpoon)". The window displays the output of the command \$ git config --list --local. The output shows various Git configuration settings, including repository format version, file mode, bare status, log all ref updates, symbolic links, ignore case, hidden dot files, remote origin URL, fetch settings, branch master remote, merge strategy, upstream URL, fetch settings, branch TeaSpoon remote, and merge strategy.

```
Nando@LIAN /C/Repos/Spoon-Knife (TeaSpoon)
$ git config --list --local
core.repositoryformatversion=0
core.filemode=false
core.bare=false
core.logallrefupdates=true
core.symlinks=false
core.ignorecase=true
core.hidedotfiles=dotGitOnly
remote.origin.url=https://github.com/fsantacroce/Spoon-Knife.git
remote.origin.fetch=+refs/heads/*:refs/remotes/origin/*
branch.master.remote=origin
branch.master.merge=refs/heads/master
remote.upstream.url=https://github.com/octocat/Spoon-Knife.git
remote.upstream.fetch=+refs/heads/*:refs/remotes/upstream/*
branch.TeaSpoon.remote=origin
branch.TeaSpoon.merge=refs/heads/TeaSpoon

Nando@LIAN /C/Repos/Spoon-Knife (TeaSpoon)
$
```

Editing configuration files manually

Even if it is generally discouraged, you can modify Git configurations by directly editing the files. Git configuration files are quite easy to understand, so when you look on the Internet for a particular configuration you want to set, it is not unusual to find just the right corresponding text lines; the only little foresight to maintain in those cases is that you always need to back up files before editing them. In the next paragraphs, we will try to make some changes in this manner.

Setting up other environment configurations

Using Git can be a painful experience if you are not able to place it conveniently inside your work environment. Let's start to shape some rough edges using a bunch of custom configurations.

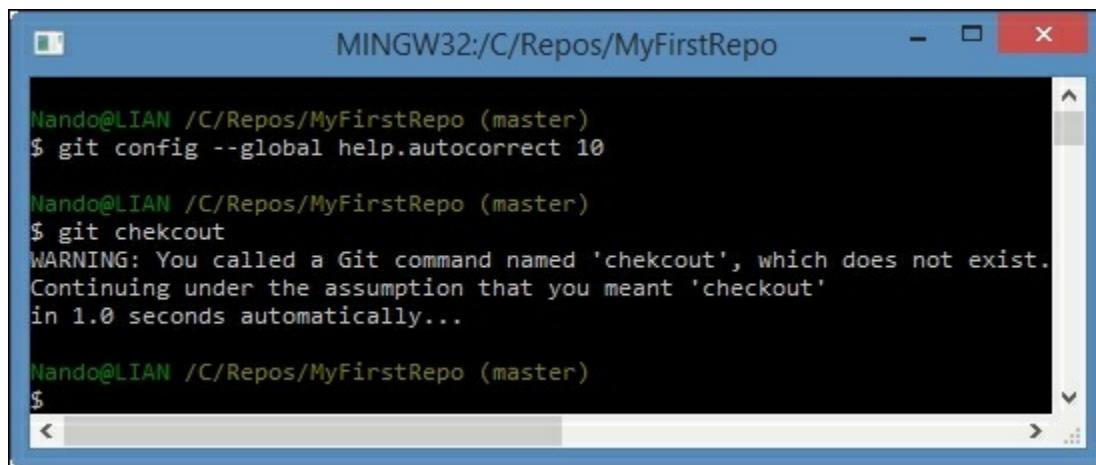
Basic configurations

In the previous chapters, we saw that we can change a Git variable value using the `git config` command with the `<variable.name>`

<value> syntax. In this paragraph, we will make use of the `config` command to vary some Git behaviors.

Typo autocorrection

So, let's try to fix an annoying question about the typing command named **typos**. I often find myself re-typing the same command two or more times; Git can help us with embedded **autocorrection**, but we first have to enable it. To enable it, you have to modify the `help.autocorrection` parameter, defining how many tenths of a second Git will wait before running the assumed command; so by giving a `help.autocorrect 10` command, Git will wait for a second, as shown in the following screenshot:

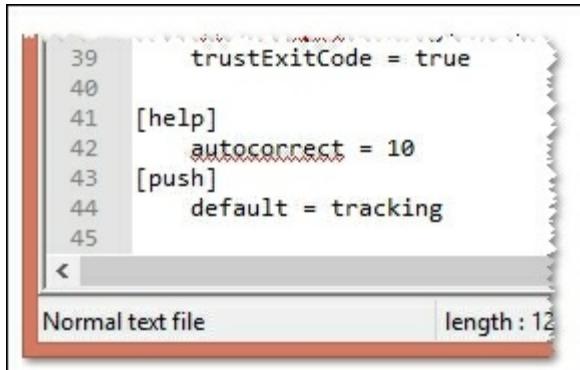


The screenshot shows a terminal window titled "MINGW32:/C/Repos/MyFirstRepo". The user has run the command `$ git config --global help.autocorrect 10`. In the next line, they type `$ git chekcout`, which is corrected by Git to `checkout`. A warning message appears: "WARNING: You called a Git command named 'chekcout', which does not exist. Continuing under the assumption that you meant 'checkout' in 1.0 seconds automatically...". The user then presses the Enter key again.

To abort the autocorrection, simply type *Ctrl + C*.

Now that you know about configuration files, you can note that the parameters we set by the command line are in this form:

`section.parameter_name`. You can see the sections' names within [] if you look in the configuration file; for example, you can find them in `C:\Users\<UserName>\.gitconfig`, as shown in the following screenshot:

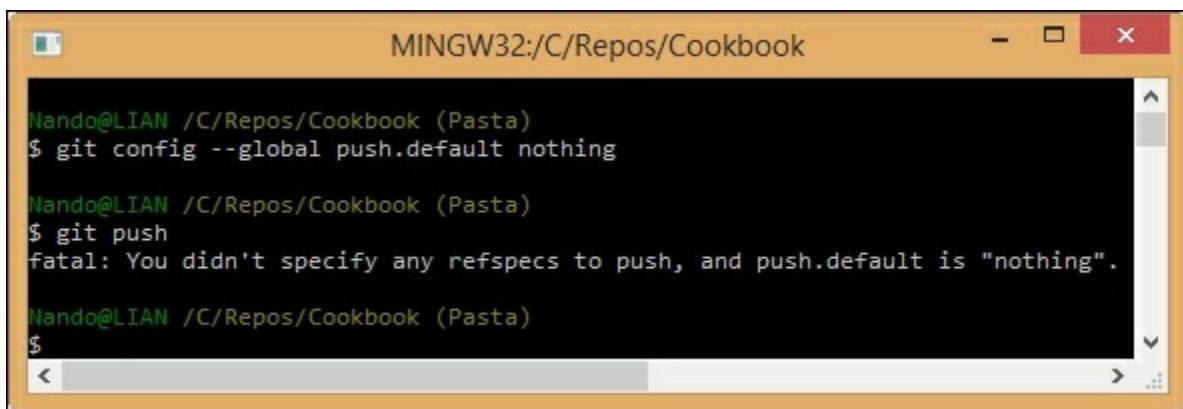


```
39     trustExitCode = true
40
41 [help]
42     autocorrect = 10
43 [push]
44     default = tracking
45
<
Normal text file length : 12
```

Push default

We already talked about the `git push` command and its default behavior. To avoid such annoying issues, it is a good practice to set a more convenient default behavior for this command.

There are two ways we can do this. The first one is to set Git to ask to us the name of the branch we want to push every time, so a simple `git push` will have no effects. To obtain this, set `push.default` to `nothing`, as shown in the following screenshot:



```
Nando@LIAN /C/Repos/Cookbook (Pasta)
$ git config --global push.default nothing

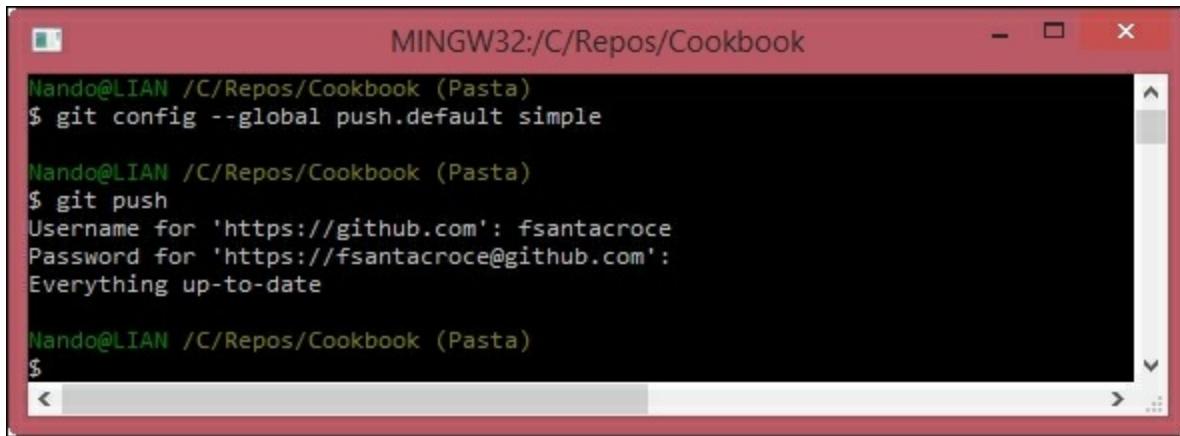
Nando@LIAN /C/Repos/Cookbook (Pasta)
$ git push
fatal: You didn't specify any refs to push, and push.default is "nothing".

Nando@LIAN /C/Repos/Cookbook (Pasta)
$
```

As you can see, now Git pretends that you specify the target branch at every push.

This may be too restrictive, but at least you can avoid common mistakes like pushing some personal local branches to the remote, generating confusion in the team.

Another way to save yourself from this kind of mistake is to set the `push.default` parameter to `simple`, allowing Git to push only when there is a remote branch with the same name as that of the local one, as shown in the following screenshot:



The screenshot shows a terminal window titled "MINGW32:/C/Repos/Cookbook". The command \$ git config --global push.default simple is run, followed by \$ git push, which prompts for a GitHub username and password. The output shows "Everything up-to-date", indicating a successful push.

```
Nando@LIAN /C/Repos/Cookbook (Pasta)
$ git config --global push.default simple

Nando@LIAN /C/Repos/Cookbook (Pasta)
$ git push
Username for 'https://github.com': fsantacroce
Password for 'https://fsantacroce@github.com':
Everything up-to-date

Nando@LIAN /C/Repos/Cookbook (Pasta)
$
```

This action will push the local tracked branch to the remote.

Defining the default editor

Some people really don't like `vim`, even only for writing commit messages; if you are one of those people, there is good news for you in that you can change it instead by setting the `core.editor` config parameter:

```
$ git config --global core.editor notepad
```

Obviously you can set all text editors on the market. If you are a Windows user, remember that the full path of the editor has to be in the `PATH` environment variable; basically, if you can run your preferred editor typing its executable name in a DOS shell, you can use it even in a Bash shell with Git.

Other configurations

You can browse a wide list of other configuration variables at <http://git-scm.com/docs/git-config>.

Git aliases

In [Chapter 2, Git Fundamentals – Working Locally](#) we already mentioned Git aliases and their purpose; in this paragraph, I will suggest only a few more to help you make things easier.

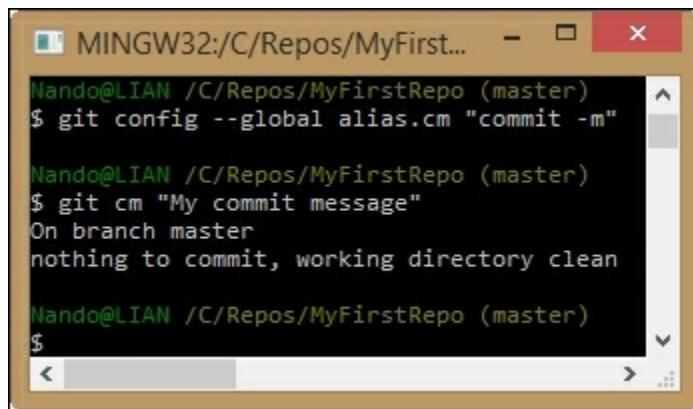
Shortcuts to common commands

One thing you can find useful is to shorten common commands like `git checkout` and so on; therefore, these are some useful aliases:

```
$ git config --global alias.co checkout  
$ git config --global alias.br branch  
$ git config --global alias.ci commit  
$ git config --global alias.st status
```

Another common practice is to shorten a command by adding one or more options that you use all the time; for example set a `git cm <commit message>` command shortcut to alias `git commit -m <commit message>`:

```
$ git config --global alias.cm "commit -m"
```



The screenshot shows a terminal window titled 'MINGW32:/C/Repos/MyFirst...'. It contains the following text:

```
Nando@LIAN /C/Repos/MyFirstRepo (master)  
$ git config --global alias.cm "commit -m"  
  
Nando@LIAN /C/Repos/MyFirstRepo (master)  
$ git cm "My commit message"  
On branch master  
nothing to commit, working directory clean  
  
Nando@LIAN /C/Repos/MyFirstRepo (master)  
$
```

Creating commands

Another common way to customize the Git experience is to create commands you think should exist, as we did in [Chapter 2, Git Fundamentals – Working Locally](#) with the `git tree` command.

git unstage

The classic example is the `git unstage` alias:

```
$ git config --global alias.unstage 'reset HEAD --'
```

With this alias, you can remove a file from the index in a more meaningful way as compared to the equivalent `git reset HEAD <file>` syntax:

```
$ git unstage myfile.txt  
$ git reset HEAD myfile.txt
```

git undo

Do you want a fast way to revert the last ongoing commit? Create a `git undo` alias:

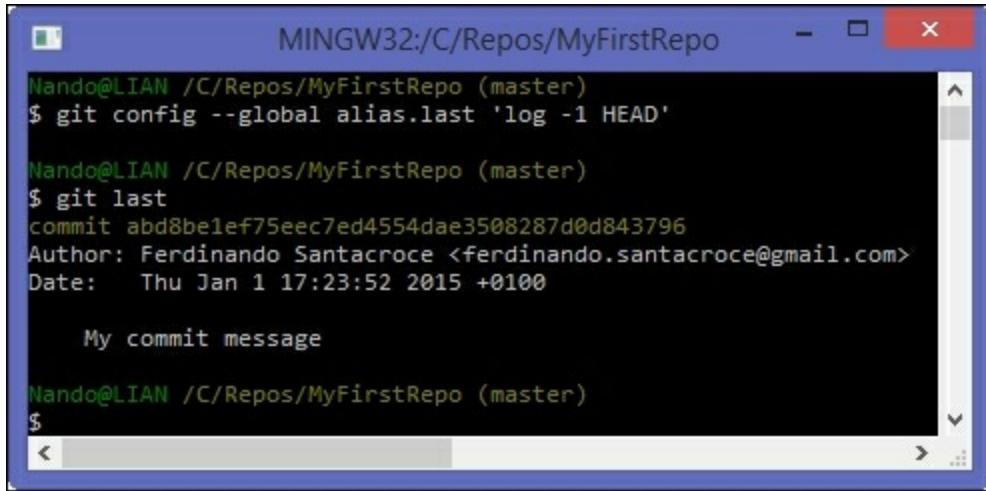
```
$ git config --global alias.undo 'reset --soft HEAD~1'
```

You will be tempted to use the `--hard` option instead of the `--soft` option, but simply don't do it as it's generally a bad idea to make it too easy to destroy information, and sooner or later, you will regret for deleting something important.

git last

A `git last` alias is useful to read about your last commit, which is shown here:

```
$ git config --global alias.last 'log -1 HEAD'
```



```
MINGW32:/C/Repos/MyFirstRepo
Nando@LIAN /C/Repos/MyFirstRepo (master)
$ git config --global alias.last 'log -1 HEAD'

Nando@LIAN /C/Repos/MyFirstRepo (master)
$ git last
commit abd8be1ef75eec7ed4554dae3508287d0d843796
Author: Ferdinando Santacroce <ferdinando.santacroce@gmail.com>
Date:   Thu Jan 1 17:23:52 2015 +0100

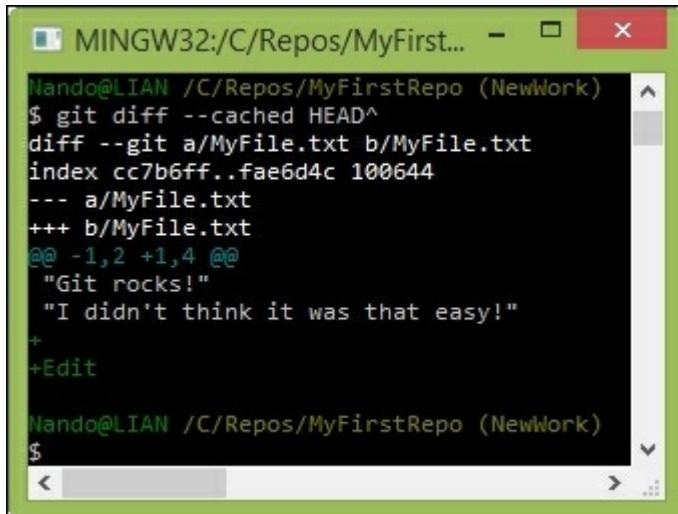
    My commit message

Nando@LIAN /C/Repos/MyFirstRepo (master)
$
```

git difflast

With `git difflast` alias, you can indeed see a difference from your last commit, as shown here:

```
$ git config --global alias.difflast 'diff --cached HEAD^'
```



```
MINGW32:/C/Repos/MyFirst...
Nando@LIAN /C/Repos/MyFirstRepo (NewWork)
$ git diff --cached HEAD^
diff --git a/MyFile.txt b/MyFile.txt
index cc7b6ff..fae6d4c 100644
--- a/MyFile.txt
+++ b/MyFile.txt
@@ -1,2 +1,4 @@
 "Git rocks!"
 "I didn't think it was that easy!"
+
+Edit

Nando@LIAN /C/Repos/MyFirstRepo (NewWork)
$
```

Advanced aliases with external commands

If you want the alias to run external shell commands instead of a Git subcommand, you have to prefix the alias with a `!:`

```
$ git config --global alias.echo !echo
```

Suppose you are annoyed by the canonical `git add <file>` plus `git commit <file>` sequence of commands, and you want to do it in a single shot; here you can call the `git` command twice in sequence creating this alias:

```
$ git config --global alias.cm '!git add -A && git commit -m'
```

With this alias you commit all the files, adding them before, if necessary.

Have you noted that I set again the `cm` alias? If you set an already configured alias, the previous alias will be overwritten.

There are also aliases that define and use complex functions or scripts, but I'll leave it to the curiosity of the reader to explore these aliases. If you are looking for inspiration, please take a look at mine at <https://github.com/jesuswasrasta/GitEnvironment>.

Removing an alias

Removing an alias is quite easy; you have to use the `--unset` option, specifying the alias to remove. For example, if you want to remove the `cm` alias, you have to run:

```
$ git config --global --unset alias.cm
```

Note that you have to specify the configuration level with the appropriate option; in this case, we are removing the alias from the user (`--global`) level.

Aliasing the git command itself

I already said I'm a bad typewriter; if you are too, you can alias the `git` command itself (using the default `alias` command in Bash):

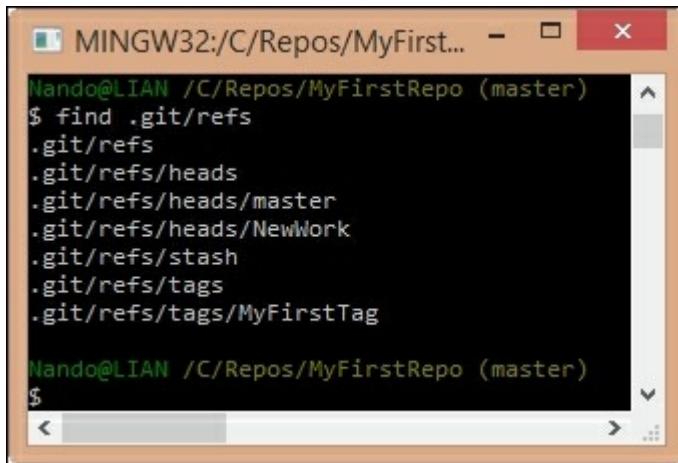
```
$ alias gti='git'
```

In this manner, you will save some other keyboard strokes. Note that this is not a Git alias but a Bash shell alias.

Git references

We said that a Git repository can be imagined as an acyclic graph, where every node, the commit, has a parent and a unique SHA-1 identifier. But during the previous chapters, we even used some references such as the `HEAD`, branches, tags, and so on.

Git manages these references as files in the `.git/refs` repository folder:



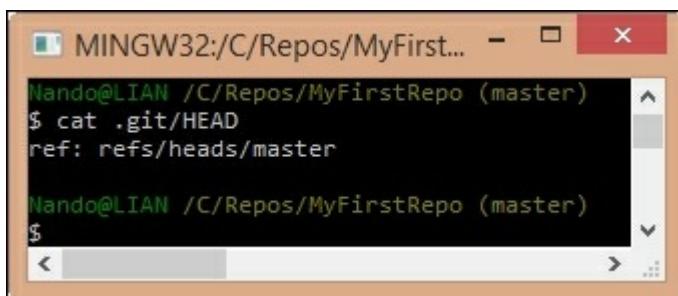
```
MINGW32:/C/Repos/MyFirst... - □ X
Nando@LIAN /C/Repos/MyFirstRepo (master)
$ find .git/refs
.git/refs
.git/refs/heads
.git/refs/stash
.git/refs/tags
.git/refs/heads/master
.git/refs/heads/NewWork
.git/refs/tags/MyFirstTag

Nando@LIAN /C/Repos/MyFirstRepo (master)
$
```

If you open one of those files, you will find it inside the SHA-1 of the commit they are tied to. As you can see, there are subfolders for tags and branches (called `heads`).

Symbolic references

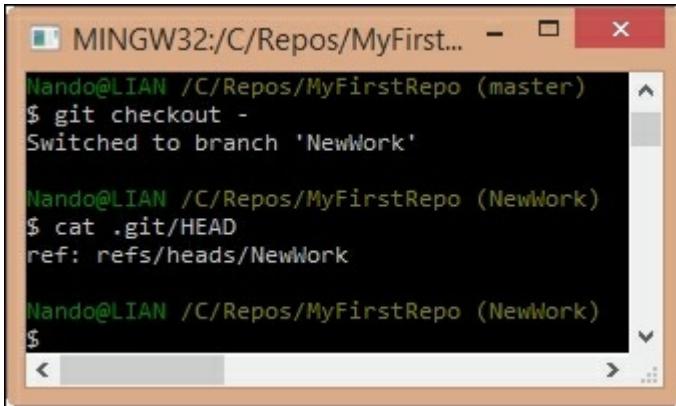
The `HEAD` file instead is located in the `.git` folder, as shown in the following screenshot:



```
MINGW32:/C/Repos/MyFirst... - □ X
Nando@LIAN /C/Repos/MyFirstRepo (master)
$ cat .git/HEAD
ref: refs/heads/master

Nando@LIAN /C/Repos/MyFirstRepo (master)
$
```

`HEAD` is a **symbolic reference**; symbolic references are references that point to other references, using the `ref: <reference>` syntax. In this case, the `HEAD` is currently pointing to the `master` branch; if you check out another branch, you will see the file's content change, as shown in the following screenshot:



A screenshot of a terminal window titled "MINGW32:/C/Repos/MyFirst...". The terminal shows the following command sequence:

```
Nando@LIAN /C/Repos/MyFirstRepo (master)
$ git checkout -
Switched to branch 'NewWork'

Nando@LIAN /C/Repos/MyFirstRepo (NewWork)
$ cat .git/HEAD
ref: refs/heads/NewWork

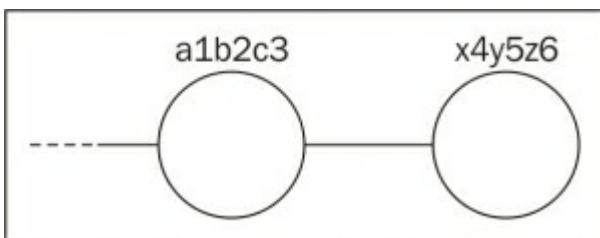
Nando@LIAN /C/Repos/MyFirstRepo (NewWork)
$
```

Ancestry references

In Git you often need to reference the past (for example, the last commit); for this scope, we can use two different special characters which are the **tilde** `~` and the **caret** `^`.

The first parent

Suppose you want to completely delete the last **x4y5z6** commit:



A way to do this is to move the `HEAD` pointer to the **a1b2c3** commit, using the `--hard` option:

```
$ git reset --hard a1b2c3
```

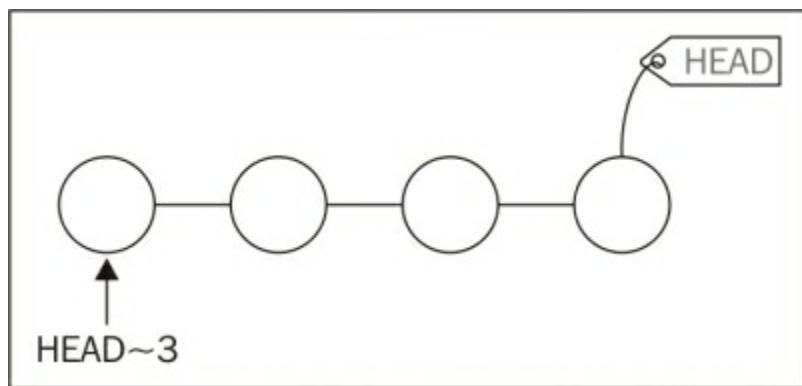
Another way to do this is to move the pointer back to the parent commit. To define the parent, you have to specify a starting point reference, which can be the `HEAD`, a specific commit, a tag, or a branch and then one of two special characters: the tilde `~`, for the first parent and the caret `^` for the second one (remember that commits can have two parents when they represent a merge result).

Let's get under the lens of the tilde `~`. With the `<ref>~<number>` notation, we can specify how many steps backward we are going to take; going back to the example, an equivalent of the previous command is this:

```
$ git reset --hard HEAD~1
```

The `HEAD~1` notation tells Git to point to the first parent commit of the actual commit (the `HEAD`, indeed). Note that `HEAD~1` and `HEAD~` are equivalent.

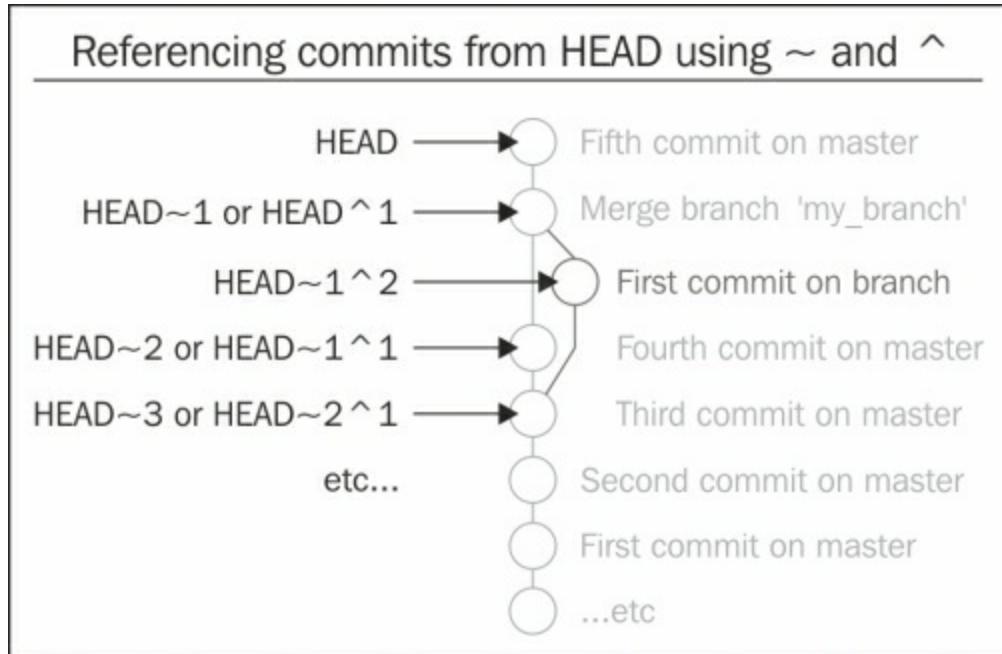
You can also go backward by more than one step, simply incrementing the number; a `HEAD~3` reference will point to the third ancestor of the `HEAD`:



The second parent

With the `^` caret character, instead we reference the second parent of a commit, but only starting from the number **2**; the `ref^1` notation references the first parent, as does the `ref~1` notation whereas `ref^` and `ref~1` are equivalent. Also note that `ref^1` and `ref^` are equivalent.

The `^` and `~` operators can be combined; here's a diagram showing how to reference various commits using `HEAD` as the starting point:



World-wide techniques

In this section, you will raise your skills by learning some techniques that will come in handy in different situations.

Changing the last commit message

This trick is for people who don't double-check what they're writing. If you pressed the *Enter* key too early, there's a way to modify the last commit message, using the `git commit` command with the `--amend` option:

```
$ git commit --amend -m "New commit message"
```

Please note that with the `--amend` option, you are actually redoing the commit, which will have a new hash; if you already pushed the previous commit, changing the last commit is not recommended; rather, it is deplorable and you will get in trouble.

Tracing changes in a file

Working on source code in a team, it is not uncommon to need to look at the last modifications made to a particular file to better understand how it evolved over time. To achieve this result, we can use the `git blame <filename>` command.

Let's try this inside the `Spoon-Knife` repository to see the changes made to the `README.md` file during that time:

The screenshot shows a terminal window titled "MINGW32:/C/Repos/Spoon-Knife". The command entered was \$ git blame README.md. The output lists the history of changes to the README.md file, showing commit hashes, authors, dates, and the specific lines affected by each commit. The commits are from "The Octocat" and date back to February 4, 2014.

```
Nando@LIAN /C/Repos/Spoon-Knife (TeaSpoon)
$ git blame README.md
bb4cc8d3 (The Octocat 2014-02-04 14:38:36 -0800 1) ### Well hello there!
bb4cc8d3 (The Octocat 2014-02-04 14:38:36 -0800 2)
bb4cc8d3 (The Octocat 2014-02-04 14:38:36 -0800 3) This repository is meant to provide an example for
bb4cc8d3 (The Octocat 2014-02-04 14:38:36 -0800 4)
bb4cc8d3 (The Octocat 2014-02-04 14:38:36 -0800 5) Creating a *fork* is producing a personal copy of s
bb4cc8d3 (The Octocat 2014-02-04 14:38:36 -0800 6)
bb4cc8d3 (The Octocat 2014-02-04 14:38:36 -0800 7) After forking this repository, you can make some ch
bb4cc8d3 (The Octocat 2014-02-04 14:38:36 -0800 8)
d0dd1f61 (The Octocat 2014-02-12 15:20:44 -0800 9) For some more information on how to fork a repositor
(END)
```

As you can see in the preceding screenshot, the result reports all the affected lines of the `README.md` file; for every line you can see the commit hash, the author, the date, and the row number of the text file lines.

Suppose now you found that the modification you are looking for is the one made in the **d0dd1f61** commit; to see what happened there, type the `git show d0dd1f61` command:

The screenshot shows a terminal window titled "MINGW32:/C/Repos/Spoon-Knife". The command entered was \$ git show d0dd1f61. The output displays the commit details (author, date, message) and a diff showing the changes made in the README.md file. The commit message is "Pointing to the guide for forking". The diff shows the addition of a section about forking the repository.

```
Nando@LIAN /C/Repos/Spoon-Knife (TeaSpoon)
$ git show d0dd1f61
commit d0dd1f61b33d64e29d8bc1372a94ef6a2fee76a9
Author: The Octocat <octocat@nowhere.com>
Date:   Wed Feb 12 15:20:44 2014 -0800

    Pointing to the guide for forking

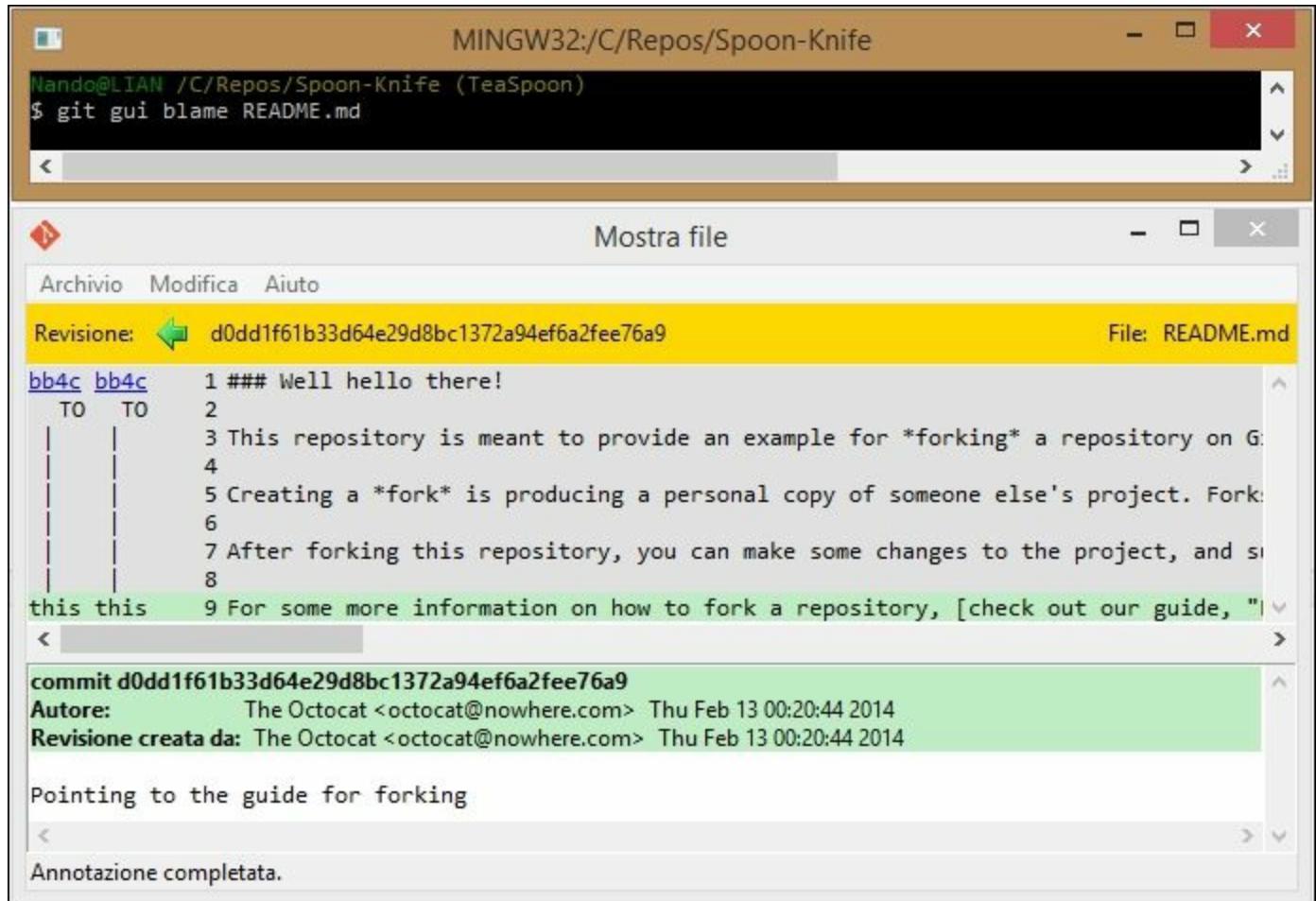
diff --git a/README.md b/README.md
index 0350da3..f479026 100644
--- a/README.md
+++ b/README.md
@@ -6,4 +6,4 @@ Creating a *fork* is producing a personal copy of someone else's proje
 After forking this repository, you can make some changes to the project, and submit [
+For some more information on how to fork a repository, [check out our guide, "Fork a +For some more information on how to fork a repository
 , [check out our guide, "Forking](END)
```

The `git show` command is a multipurpose command, it can show you one or more objects; in this case we have used it to show the modification made in a particular commit using the `git show <commit-hash>` format.

The `git blame` and `git show` commands have quite a long list of options; the purpose of this paragraph is only to point the reader to the

way changes should be traced on a file; you can inspect other possibilities using the ever useful `git <command> --help` command.

The last tip I want to suggest is to use the Git GUI:



The screenshot shows a Git GUI interface. At the top, a terminal window displays the command `git gui blame README.md`. Below it, a main window titled "Mostra file" shows the content of the README.md file. The file contains the following text:

```
bb4c bb4c 1 ### Well hello there!
TO TO 2
| | 3 This repository is meant to provide an example for *forking* a repository on G:
| | 4
| | 5 Creating a *fork* is producing a personal copy of someone else's project. Fork:
| | 6
| | 7 After forking this repository, you can make some changes to the project, and so
| | 8
this this 9 For some more information on how to fork a repository, [check out our guide, "]

commit d0dd1f61b33d64e29d8bc1372a94ef6a2fee76a9
Autore: The Octocat <octocat@nowhere.com> Thu Feb 13 00:20:44 2014
Revisione creata da: The Octocat <octocat@nowhere.com> Thu Feb 13 00:20:44 2014

Pointing to the guide for forking
Annotazione completata.
```

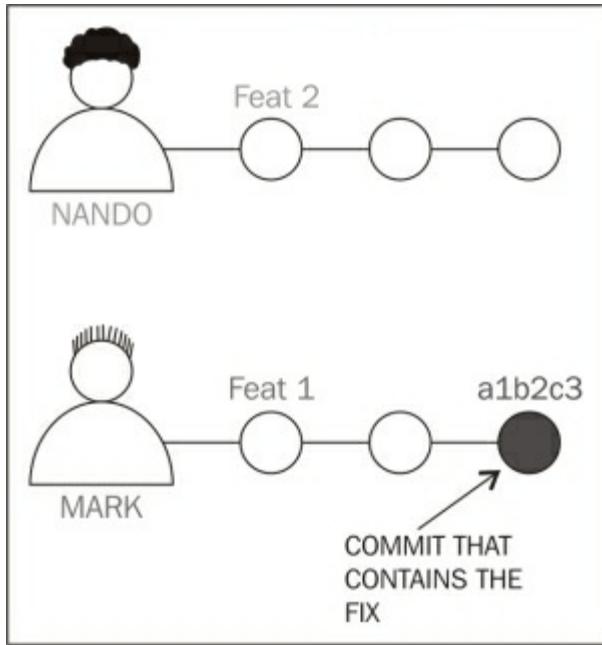
With the help of GUI, things are much more easy to understand.

Cherry picking

The **cherry picking** activity consists of choosing existing commits from somewhere else and applying them here. I will make use of an example to better explain how you can benefit from this technique.

Suppose you and your colleague Mark are working on two different public branches of the same repository; Mark found and fixed an annoying bug in the `feat1` branch that affects even your `feat2` branch. You need that fix, but you can't (or don't want to) merge his branch, so

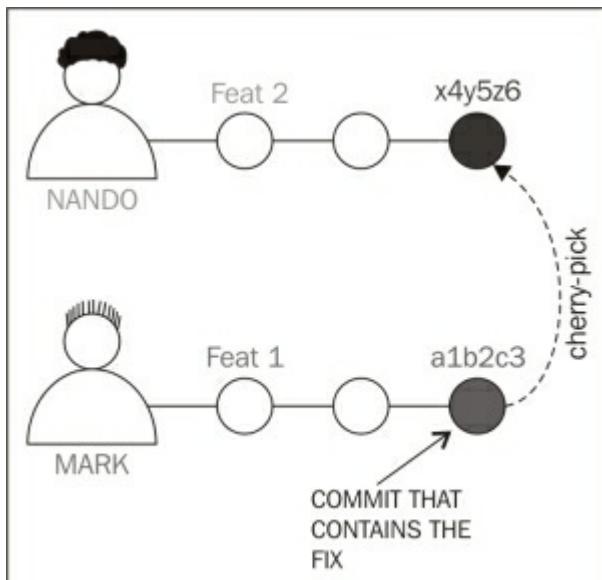
how can you benefit from his fix?



It's easy; get the commit that fixes that bug and apply it to your current branch, using the `git cherry-pick` command:

```
$ git checkout feat2  
$ git cherry-pick a1b2c3
```

That's all! Now the commit **a1b2c3** performed by Mark in the `feat1` branch has been applied to the `feat2` branch and committed as a new **x4y5z6** commit, as shown in the following screenshot:



The `git cherry-pick` command behaves just like the `git merge` command. If Git can't apply the changes (for example, you get merge conflicts), it leaves you to resolve the conflicts manually and make the commit yourself.

We can even pick commit sets if we want to by using the `<starting-commit>..<ending-commit>` syntax:

```
$ git cherry-pick feat1~2..feat1~0
```

With this syntax, you are basically picking the last two commits from the `feat1` branch.

Tricks

In this section, I would suggest just a bunch of tips and tricks that I found useful in the past.

Bare repositories

Bare repositories are repositories that do not contain working copy files but contain only the `.git` folder. A bare repository is essentially for sharing; if you use Git in a centralized way, pushing and pulling to a common remote (a local server, a GitHub repository, or so on), you will agree that the remote has no interest in checking out files you work on; the scope of that remote is only to be a central point of contact for the team, so having working copy files in it is a waste of space, and no one will edit them directly on the remote.

If you want to set up a bare repository, you have to use only the `--bare` option:

```
$ git init --bare NewRepository.git
```

As you may have noticed, I called it `NewRepository.git`, using a `.git` extension; this is not mandatory but is a common way to identify bare repositories. If you pay attention, you will note that even in GitHub every repository ends with the `.git` extension.

Converting a regular repository to a bare one

It can happen that you start working on a project in a local repository and then you feel the need to move it to a centralized server to make it available to other people or locations.

You can easily convert a regular repository to a bare one using the `git clone` command with the same `--bare` option:

```
$ git clone --bare my_project my_project.git
```

In this manner, you have a 1:1 copy of your repository, but in a bare

version, ready to be shared.

Backup repositories

If you need a backup, there are two commands you can use, one of which is for archiving only files and the other is for backing up the entire bundle including the versioning information.

Archiving the repository

To archive the repository without including the versioning information, you can use the `git archive` command; there are many output formats of which ZIP is the classic one:

```
$ git archive master --format=zip --output=../repbck.zip
```

Please note that using this command is not the same as backing up folders in the filesystem; as you noticed, the `git archive` command can produce archives in a smarter way, including only files in a branch or even in a single commit. By doing this, you are archiving only the last commit, as shown in the following code:

```
$ git archive HEAD --format=zip --output=../headbck.zip
```

Archiving files in this way can be useful if you have to share your code with people that don't have Git installed.

Bundling the repository

Another interesting command is the `git bundle` command. With `git bundle` you can export a snapshot from your repository, and you can then restore it.

Suppose you want to clone your repository on another computer, and the network is down or absent; with this command, you create a `repo.bundle` file of the `master` branch:

```
$ git bundle create ../repo.bundle master
```

With these other commands, we can then restore the bundle in the other

computer using the `git clone` command, as shown here:

```
$ cd /OtherComputer/Folder  
$ git clone repo.bundle repo -b master
```

Summary

In this chapter, you enhanced your knowledge about Git and its wide set of commands. You finally understood how configuration levels work and how to set your preferences using Git, by adding useful command aliases to the shell. Then we looked at how Git deals with references, providing a way to refer to a previous commit using its degree of relationship.

Furthermore, you added some key techniques to your skill set, as it is important to learn something you will use as soon as you start to use Git extensively. You also learned some simple tricks to help you use Git more efficiently.

Chapter 5. Obtaining the Most – Good Commits and Workflows

Now that we are familiar with Git and versioning systems, it's time to look at the whole thing from a much higher perspective to become aware of common patterns and procedures.

In this chapter, we will walk through some of the most common ways to organize and build meaningful commits and repositories. We will obtain not only a well-organized code stack, but also a meaningful source of information.

The art of committing

While working with Git, committing seems the easiest part of the job: you add files, write a short comment, and then, you're done. However, it is because of its simplicity that often, especially at the very beginning of your experience, you acquire the bad habit of doing terrible commits: too late, too big, too short, or simply equipped with bad messages.

Now, we will take some time to identify possible issues, drawing attention to tips and hints to get rid of these bad habits.

Building the right commit

One of the harder skills to acquire while programming in general is to split the work in small and meaningful tasks.

Too often, I have experienced this scenario. You start to fix a small issue in a file. Then, you see another piece of code that can be easily improved, even if it is not related to what you are working on now. You can't resist it, and you fix it. At the end and after a small time, you find yourself with tons of *concurrent* files and *changes* to commit.

At this point, things get worse, because usually, programmers are lazy

people. So, they don't write all the important things to describe changes in the commit message. In commit messages, you start to write sentences such as "Some fixes to this and that", "Removed old stuff", "Tweaks", and so on, without anything that helps other programmers understand what you have done.

COMMENT	DATE
○ CREATED MAIN LOOP & TIMING CONTROL	14 HOURS AGO
○ ENABLED CONFIG FILE PARSING	9 HOURS AGO
○ MISC BUGFIXES	5 HOURS AGO
○ CODE ADDITIONS/EDITS	4 HOURS AGO
○ MORE CODE	4 HOURS AGO
○ HERE HAVE CODE	4 HOURS AGO
○ AAAAAAAA	3 HOURS AGO
○ ADKFJSLKDFJSOKLFJ	3 HOURS AGO
○ MY HANDS ARE TYPING WORDS	2 HOURS AGO
○ HAAAAAAAAANDS	2 HOURS AGO

AS A PROJECT DRAGS ON, MY GIT COMMIT MESSAGES GET LESS AND LESS INFORMATIVE.

Courtesy of <http://xkcd.com/1296/>

At the end, you realize that your repository is only a dump where you empty your index now and then. I have seen some people committing only at the end of the day (and not every day) to keep a backup of the data or because someone else needed the changes reflected on their computer.

Another side effect is that the resulting repository history becomes useless for anything other than retrieving content at a given point in time.

The following tips can help you turn your **Version Control System (VCS)** from a backup system into a valuable tool for communication and documentation.

Make only one change per commit

After the routine morning coffee, you open your editor and start to work on a bug, BUG42. While working around fixing the bug in the code, you

realize that fixing BUG79 will require tweaking just a single line of code. So, you fix it. However, you not only change that awful class name, but also add a good-looking label to the form and make a few more changes. The damage is done.

How can you wrap up all that work in a meaningful commit now? Maybe, in the meantime, you went home for lunch, talked to your boss about another project, and you can't even remember all the little things you did.

In this scenario, there is only one way to limit the damage: split the files to commit among more than one commit. Sometimes, this helps to reduce the pain, but it is only palliative. Very often, you modify the same file for different reasons, so doing this is quite difficult, if not impossible. The last hope is to use `git add -p` command, that let's you to stage only some modification on a file, grouping them in different commit to separate topics.

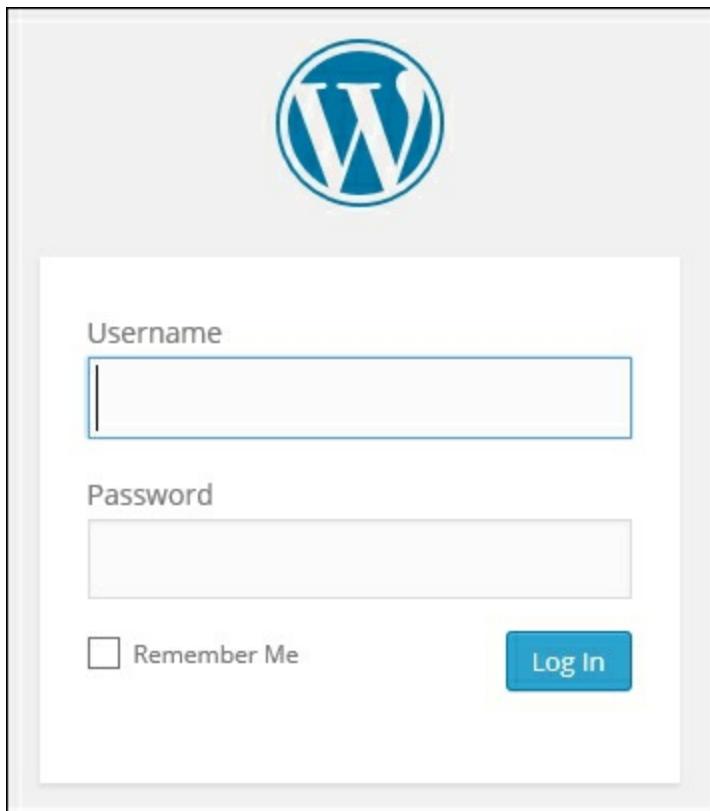
The only way to definitely solve this problem is to only make one change per commit. It seems easy, I know, but it is quite difficult to acquire this ability. There are no tools for this. No one, but you, can help. It only needs discipline, the most lacking virtue in creative people such as programmers.

There are some tips to pursue this aim; let's have a look at them together.

Split up features and tasks

As said earlier, breaking up the things to do is a fine art. If you know and adopt some **Agile movement** techniques, you will have probably faced these problems. So, you have an advantage; otherwise, you will need some more effort, but it is not something that you can't achieve.

Consider that you have been assigned to add the **Remember me** check in the login page of a web application, like the one shown here:



This feature is quite small, but implies changes at different levels. To accomplish this, you'll have to:

- Modify the UI to add the check control
- Pass the "is checked" information through different layers
- Store this information somewhere
- Retrieve this information when needed
- Invalidate (set it to false) following some kind of policy (after 15 days, after 10 logins, and so on)

Do you think you can do all these things in one shot? Yes? You are wrong! Even if you estimate a couple of hours for an ordinary task, remember that Murphy's law is in ambush. You will receive four calls, your boss will look for you for three different meetings, and your computer will go up in flames.

This is one of the first things to learn: break up every work into small tasks. It does not matter whether you use timeboxing techniques such as the **Pomodoro Technique**; small things are always easy to handle. I'm

not talking about split hairs, but try to organize your tasks into things you can do in a defined amount of time, hopefully a bunch of half hours, not days.

So, take a pen and paper and write down all the tasks, as we did earlier with the login page example. Do you think you can do all those things in a small amount of time? Maybe yes, maybe not: some tasks are bigger than others. That's OK; this is not a scientific method. It's a matter of experience. Can you split a task and create two other meaningful tasks? Do it.

Are you unable to do it? No problem; don't try to split tasks if they lose meaning.



Write commit messages before starting to code

Now, you have a list of tasks to do; pick the first and... start to code? No! Take another piece of paper and describe every task's step with a sentence. Magically, you will realize that every sentence can be the message of a single commit, where you describe the features you deleted, added, or changed in your software.

This kind of prior preparation helps you define modifications to implement (letting better software design to emerge). It also focuses on what is important and lowers down the stress of thinking at the versioning part of the work during the coding session. While you are

facing a programming problem, your brain floods with little implementation details related to the code you are working on. So, the fewer the distractions, the better.

This is one of the best versioning-related hints I ever received. If you have just a quarter of an hour to spare, I recommend that you read the *Preemptive commit comments* blog post (<https://arialdomartini.wordpress.com/2012/09/03/pre-emptive-commit-comments/>) by *Arialdo Martini*. This is where I learnt this trick.

Include the whole change in one commit

Making more than one change per commit is a bad thing. However, splitting a single change into more than one commit is also considered harmful. As you may know, in some trained teams, you do not simply push your code to production. Before that, you have to pass some code quality reviews, where someone else tries to understand what you did to decide if your code is good or not (that is, why there are pull requests, indeed). You could be the best developer in the world. However, if the person at the other end can't get a sense of your commits, your work would probably be refused.

To avoid these unpleasant situations, you have to follow a simple rule: don't do partial commits. If time's up, if you have to go to that damn meeting (programmers hate meetings) or whatever, remember that you can save your work at any moment without committing, using the `git stash` command. If you want to close the commit, because you want to push it to the remote branch for backup purposes, remember that *Git is not a backup tool*. Back up your stash on another disk, put it in the cloud, or simply end your work before leaving, but don't do commits like they are episodes of a TV series.

One more time, Git is a software tool like any other and it can fail. Don't think that just because you are using Git or other versioning systems, you don't need backup strategies. Back up local and remote repositories just like you back up all the other important things.

Describe the change, not what you have done

Too often, I read (and often I wrote) commit messages such as "Removed this", "Changed that", "Added that one", and so on.

Imagine that you are going to work on the common "lost password" feature on your website. Probably, you will find a message like this adequate: "Added the lost password retrieval link to the login page". This kind of commit message does not describe what modifications the feature brings to you, but what you did (and not everything). Try to answer sincerely. If you are reading a repository history, do you want to read what every developer did? Or is it better to read the feature implemented in every single commit?

Try to make the effort, and start writing sentences where the change itself is the subject, not what you did to implement it. Use the imperative present tense (for example, fix, add, implement), describing the change in a small subject sentence, and then, add some details (when needed) in other lines of text. "Implement the password-retrieval mechanism" is a good commit message subject. If you find it useful, then you can add some other information to get a well formed message like this:

"Implement the password retrieval mechanism

- Add the "Lost password?" link into the login page - Send an email to the user with a link to renew the password"

Have you ever written a changelog for a software by hand? I did; it's one of the most boring things to do. If you don't like writing changelogs, like me, think of the repository history as your changelog. If you take care of your commit messages, you would get a beautiful changelog for free!

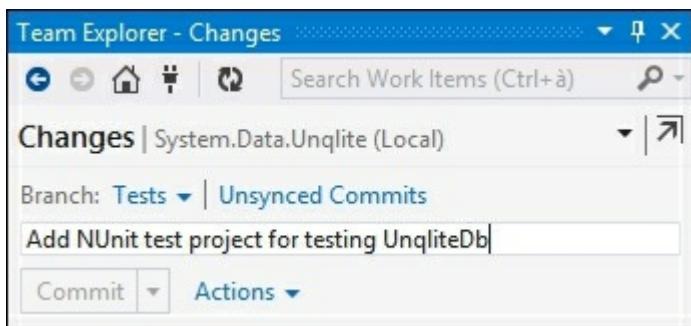
In the next section, I will group some other useful hints about good commit messages.

Don't be afraid to commit

Fear is one of the most powerful emotions. It can drive a person to do the craziest thing on Earth. One the most common reactions to fear is breakdown. You don't know what to do, so you end up doing nothing.

This is a common reaction even when you begin to use a new tool such as Git, where gaining confidence can be difficult. For the fear of making a mistake, you don't commit until you are obligated. This is the real mistake; be scared. In Git, you don't have to be scared. Maybe the solution is not obvious; maybe you have to dig on the Internet to find the right way. However, you can get off with small or no consequences, ever (well, unless you are a hard user of the `--hard` option).

On the contrary, you have to make the effort to commit often, as soon as possible. The more frequently you commit, the smaller are your commits; the smaller are your commits, the easier it is to read and understand the changelog. It is also easier to cherry-pick commits and do code reviews. To help myself get used to committing this way, I followed this simple trick: write the commit message in Visual Studio before starting to write any code.



Try to do the same in your IDE or directly in the Bash shell; it helps a lot.

Isolate meaningless commits

The golden rule is to avoid meaningless commits. However, sometimes, you need to commit something that is not a real implementation, but only a cleanup, such as deleting old comments, formatting rearrangement, and so on.

In these cases, it is better to isolate this kind of code change in separate commits. By doing this, you prevent another team member from running towards you with a knife in his hand, frothing at the mouth. Don't

commit meaningless changes and mix up them with real ones. Otherwise, other developers (and you, after a couple of weeks) will not understand them while diffing.

The perfect commit message

Let me be honest; the perfect message does not exist. If you work alone, you will probably find the best way for you. However, when in a team, there are different minds and different sensibilities, so what is good for me may not be as good for another.

In this case, you have to sit around a table and discuss. You should try to end up with a shared standard that probably would not be the one you prefer, but at least is a way to start a common path.

Rules for a good commit message really depend on the way you and your team work every day, but some common hints can be applied by everyone. They are described in the following sections.

Writing a meaningful subject

The subject of a commit is the most important part; its role is to make clear what the commit contains. Avoid technical details of other things that a common developer can understand on opening the code. Focus on the big picture. Remember that every commit is a sentence in the repository history. So, wear the hat of the changelog reader and try to write the most convenient sentence for him, not for you. Use the present tense, and write a sentence with a maximum of 50 characters.

A good subject is one like this, "Add the newsletter signup in homepage".

As you can see, I used the imperative past tense. More importantly I didn't say what I have done, but what the feature does: it added a newsletter signup box to my website.

The 50 char rule is due to the way you use Git from the shell or GUI tools. If you start to write long sentences, reviewing logs and so on can

become a nightmare. So, don't try to be the Stephen King of commit messages. Avoid adjectives and go straight to the point. You can then write additional details lines.

Another thing to remember is to start with capital letters. Do not end sentences with periods; they are useless and even dangerous.

Adding bulleted details lines, when needed

Often, you can't say all that you want in 50 chars. In this case, use details lines. In this situation, the common rule is to leave a blank line after the subject, use a dash, and go no longer than 72 chars:

```
"Add the newsletter signup in homepage
```

- Add textbox and button on homepage
- Implement email address validation
- Save email in database"

In these lines, add a few details, but not too many. Try to describe the original problem (if you fixed it) or the original need, why these functionalities have been implemented (what problem solves), and understand the possible limitations or issues.

Tie other useful information

If you use some issue and project-tracking systems, write down the issue number, bug IDs or everything else that helps:

```
"Add the newsletter signup in homepage
```

- Add textbox and button on homepage
- Implement email address validation
- Save email in database

```
#FEAT-123: closed"
```

Special messages for releases

Another useful thing is to write special format commit messages for releases so that it will be easier to find them. I usually decorate subjects with some special characters, but nothing more. To highlight a particular

commit, such as a release one, there is the `git tag` command, remember?

Conclusions

At the end, my suggestion is to try to compose your personal commit message standard by following previous hints, looking at message strategies adopted by great projects and teams around the Web, but especially by doing it. Your standard will change for sure as you evolve as a software developer and Git user. So, start as soon as possible, and let time help you find the perfect way to write a commit message.

At least, don't imitate them: <http://www.commitlogsfromlastnight.com>.

Adopting a workflow – a wise act

Now that you learned how to perform good commits, it's time to fly higher and think about **workflows**. Git is a tool for versioning, but as with other powerful tools, like knives, you can cut tasty sashimi or relieve yourself of some fingers.

The things that separate a great repository from a junkyard are the way you manage releases, the way you react when there is a bug to fix in a particular version of your software, and the way you act when you have to make users beta-test the incoming features.

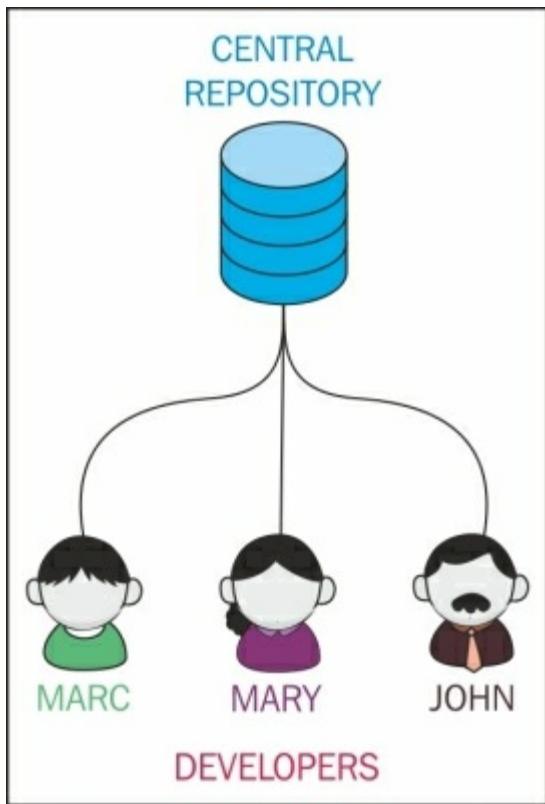
These kinds of actions belong to ordinary administration for a modern software project. However, very often, I still see teams get out of breath because of the poor versioning workflows.

In this second part of the chapter, we will take a quick look at some of the common workflows alongside the Git versioning system.

Centralized workflows

As we used to do in other VCSes, such as Subversion and so on, even in Git, it is common to adopt a *centralized* way of work. If you work in a team, it is often necessary to share repositories with others, so a common point of contact becomes indispensable.

We can assume that if you are not alone in your office, you would adopt one of the variations of this workflow. As we know, we can get all the computers of our co-workers as remote, in a sort of peer-to-peer configuration. However, you usually don't do this, because it becomes too difficult to keep every branch in every remote in sync.



How they work

In this scenario, you usually follow these simple steps:

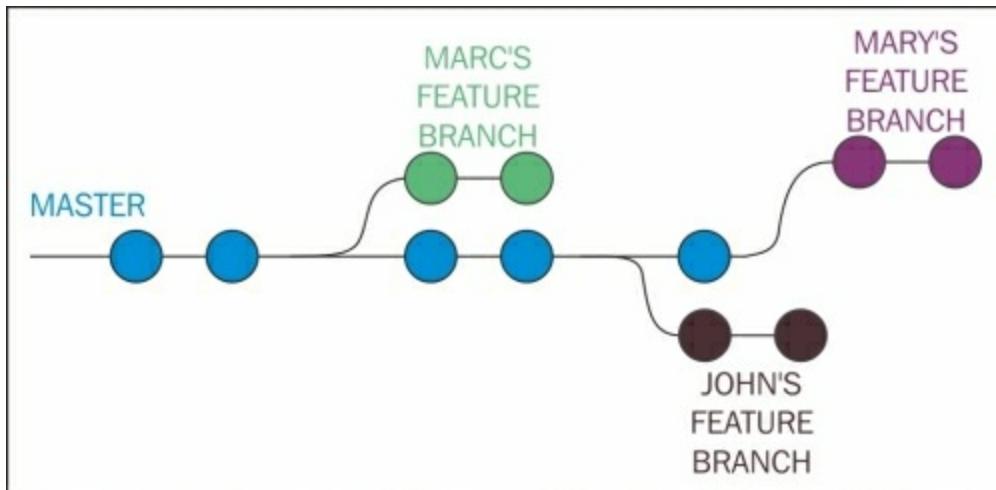
1. Someone initializes the remote repository (in a local Git server, on GitHub, or on Bitbucket).
2. Other team members clone the original repository on their computer and start working.
3. When the work is done, you push it to the remote to make it available to other colleagues.

At this point, it is only a matter of internal rules and patterns. It is improbable that you and your colleague will work together simultaneously in the `master` branch, unless you are indomitable masochists.

Feature branch workflow

At this point, you probably will choose a *feature branch* approach, where every single developer works on their branch. When the work is

done, the feature branch is ready to be merged with the master branch. You will probably have to merge back from the master branch first because one of your other colleagues has merged a feature branch after you started yours, but after that you basically have finished.

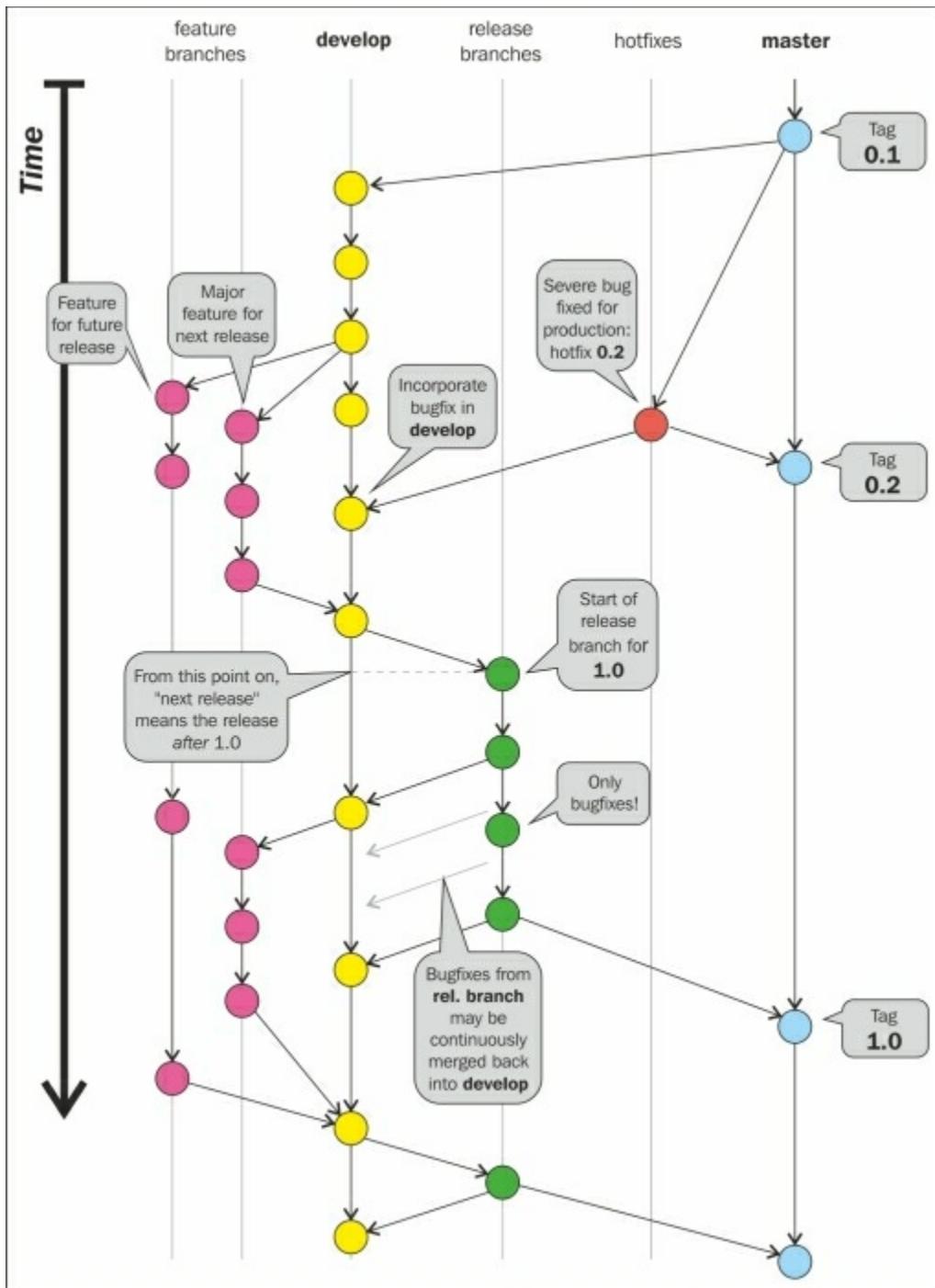


GitFlow

The **GitFlow** workflow comes from the mind of Vincent Driessen, a passionate software developer from the Netherlands. You can find his original blog post at <http://nvie.com/posts/a-successful-git-branching-model>.

His workflow has gained success over the years, at the point that many other developers (including me), teams and companies started to use it. Atlassian, a well-known company that offers Git related services such as Stash or Bitbucket, integrates the GitFlow directly in its GUI tool, the SourceTree.

Even the GitFlow workflow is a centralized one, and it is well described by this figure:



This workflow is based on the use of some **main branches**. What makes these branches special is nothing other than the significance we attribute to them. These are not special branches with special characteristics in Git, but we can certainly use them for different purposes.

The master branch

In GitFlow, the `master` branch represents the final stage. Merging your

work in it is equal to making a new release of your software. You usually don't start new branches from the `master` branch. You do it only if there is a severe bug you have to fix instantly, even if that bug has been found and fixed in another evolving branch. This way to operate makes you superfast when you have to react to a painful situation. Other than this, the `master` branch is where you tag your release.

Hotfixes branches

Hotfixes branches are branches derived only from the `master` branch, as we said earlier. Once you have fixed a bug, you merge the `hotfix` branch onto `master` so that you get a new release to ship. If the bug has not been resolved anywhere else in your repository, the strategy would be to merge the `hotfix` branch even into the `develop` branch. After that, you can delete the `hotfix` branch, as it has hit the mark.

In Git, there is a trick to group similar branches: you have to name them using a common prefix followed by a slash /. For the `hotfix` branches, I recommend the `hotfix/<branchName>` prefix (for example `hotfix/LoginBug` or `hotfix/#123` for those who are using bug-tracking systems, where #123 is the bug ID).

These branches are usually not pushed to remote. You push them only if you need the help of other team members.

The develop branch

The `develop` branch is a sort of *beta software* branch. When you start to implement a new feature, you have to create a new branch starting from the `develop` branch. You will continue to work in that branch until you complete your task.

After the task is completed, you can merge back to the `develop` branch and delete your `feature` branch. Just like `hotfix` branches, these are only temporary branches.

Like the `master` branch, the `develop` branch is a never-ending branch. You will never close nor delete it.

This branch is pushed and shared to a remote Git repository.

The release branch

At some point, you need to wrap up the next release, including some of the features you implemented in the last few weeks. To prepare an incoming release, you have to branch from `develop`, assigning at the branch a name composed of the `release` prefix. This will be followed by the numeric form of your choice for your `release` branch (for example `release/1.0`).

Pay attention. At this stage, no more new features are allowed! You cannot merge `develop` onto the `release` branch. You can only create new branches from that branch for bug fixing. The purpose of this intermediate branch is to give the software to beta testers, allowing them to try it and send you feedback and bug tickets.

If you have fixed some bugs onto the `release` branch, the only thing to remember is to merge them even into the `develop` branch, just to avoid the loss of the bug fix. The `release` branch will not be merged back to `develop`.

You can keep this branch throughout your life, until you decide that the software is both mature and tested sufficiently to go in production. At this point, you merge the `release` branch onto the `master` branch, making, in fact, a new release.

After the merge to `master`, you can make a choice. You could keep the `release` branch open, if you need to keep alive different releases; otherwise, you can delete it. Personally, I always delete the `release` branch (as Vincent suggests), because I generally do frequent, small, and incremental releases (so, I rarely need to fix an already shipped release). As you certainly remember, you can open a brand new branch from a commit (a tagged one in this case) whenever you want. So, at most, I will open it from that point only when necessary.

This branch is pushed and shared to a common remote repository.

The feature branches

When you have to start the implementation of a new feature, you have to create a new branch from the `develop` branch. Feature branches start with the `feature/` prefix (for example, `feature/NewAuthenitcation` or `feature/#987` if you use some feature- tracking software, as #987 is the feature ID).

You will work on the feature release until you finish your work. I suggest that you frequently merge back from `develop`. In the case of concurrent modifications to the same files, you will resolve conflicts faster if you resolve them earlier. Then, it is easier to resolve one or two conflicts a time than dozens at the end of the feature work.

Once your work is done, you merge the feature onto `develop` and you are done. You can now delete the `feature` branch.

Feature branches are mainly private branches. However, you could push them to the remote repository if you have to collaborate on it with some other team mates.

Conclusion

I recommend that you take a look at this workflow, as I can assure you that there was never a situation that I failed to solve using this workflow.

You can find a deeper explanation with the ready-to-use Git command on Vincent Driessen's blog. You can even use GitFlow commands Vincent made to customize his Git experience. Check them out on his GitHub account at <https://github.com/nvie/gitflow>.

The GitHub flow

The previously described GitFlow has tons of followers, but it is always a matter of taste. Someone else found it too complex and rigid for their situation. In fact, there are other ways to manage software repositories that have gained consensus during the last few years.

One of these is the workflow used at GitHub for internal projects and repositories. This workflow takes the name of **GitHub flow**. It was first described by the well-known Scott Chacon, former GitHubber and *ProGit* book author, on his blog at <http://scottchacon.com/2011/08/31/github-flow.html>.

This workflow, compared to GitFlow, is better tailored for frequent releases. When I say frequent, I say very frequently, even twice a day. Obviously, this kind of flow works better on web projects, because to deploy it, you have to *only* put the new release on the production server. If you develop desktop solutions, you need a perfect oiled update mechanism to do the same.

GitHub software basically doesn't have releases, because they deploy to production regularly, even more than once a day. This is possible due to a robust **Continuous Delivery** structure, which is not so easy to obtain. It deserves some effort.

The GitHub flow is based on these simple rules.

Anything in the master branch is deployable

Just like GitFlow, even in GitHub flow, deployment is done from the `master` branch. This is the only main branch in this flow. In GitFlow, there are not `hotfix`, `develop`, or other particular branches. Bug fixes, new implementation, and so on are constantly merged onto the `master` branch.

Other than this, code in the `master` branch is always in a deployable state. When you fix or add something new in a branch and then merge it onto the `master` branch, you don't deploy automatically, but you can assume your changes will be up and running in a matter of hours.

Branching and merging constantly to the `master` branch, which is the production-ready branch, can be dangerous. You can easily introduce regressions or bugs, as no one other than you can assure you have done a good job. This problem is avoided by a social contract commonly

adopted by GitHub developers. In this contract, you promise to test your code before merging it to the `master` branch, assuring that all automated tests have been successfully completed.

Creating descriptive branches off of the master

In GitFlow, you always branch from the `master` branch. So, it's easy to get a forest of branches to look at when you have to pull one. To better identify them, in GitHub flow, you have to use descriptive names to get meaningful topic branches. Even here, it is a matter of good manners. If you start to create branches named `stuff-to-do`, you would probably fail in adopting this flow. Some examples are `new-user-creation`, `most-starred-repositories`, and so on (note the use of dashes). Using a common way to define topics, you will easily find branches you are interested in, looking for topics' keywords.

Pushing to named branches constantly

Another great difference between GitHub flow and GitFlow is that in GitHub flow, you push feature branches to the remote regularly, even if you are the only developer involved and interested in it. This is done even for backup purposes. Even if I already exposed my opinion in merit, I can't say this is a bad thing.

A thing I appreciate about GitFlow is that this habit of pushing every branch to the remote gives you the ability to see, with a simple `git fetch` command, all the branches currently active. Due to this, you can see all the work in progress, even that of your team mates.

Opening a pull request at any time

In [Chapter 3, Git Fundamentals – Working Remotely](#), we talked about GitHub and made a quick try with pull requests. We have seen that basically they are for contributing. You fork someone else's repository, create a new branch, make some modifications, and then ask for a pull request from the original author.

In GitHub flow, you use pull requests massively. You can even ask

another developer of your team to have a look at your work and help you, give you a hint, or review the work done. At this point, you can start a discussion about using the GitHub pull request to chat and involve other people, putting their usernames in CC. In addition, the pull request feature lets you comment even a single line of code in a different view, letting users involved proficiently discuss the work under revision.

Merging only after a pull request review

You can now understand that the pull requested branch stage we saw earlier becomes a sort of review stage. Here, other users can take a look at the code and even simply leave a positive comment, just a +1 to make other users know that they are confident about the job, and they approve its merge into `master`.

After this step, when the CI server says that the branch still passes all the automated tests, you are ready to merge the branch in `master`.

Deploying immediately after review

At this stage, you merge your branch into `master`, and the work is done. The deployment is not instantly fired, but at GitHub, they have a very straight and robust deploy procedure. They deploy big branches with 50 commits, but also branches with a single commit and a single line of code change, because deployment is very quick and cheap for them.

This is the reason why they can afford such a simple branching strategy, where you put on the `master` branch, and then you deploy, without the need to pass through the `develop` or `release` stage branch, like in GitFlow.

Conclusions

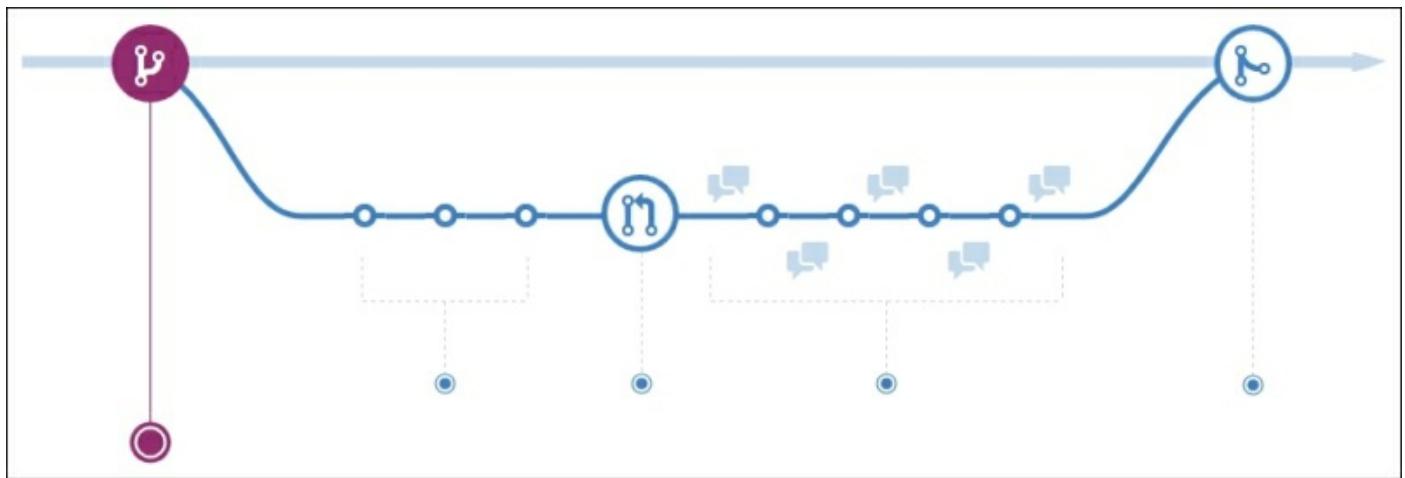
I consider this flow very responsive and effective for web-based projects, where basically you deploy to production without focusing too much on versions of your software. Using only the `master` branch to derive and integrate branches is faster than light. However, this strategy

could be applied only if you have these prerequisites:

- A centralized remote ready to manage pull requests (as GitHub does)
- A good shared agreement about branch names and pull requests usage
- A very robust deploy system

This is a big picture of this flow. For more details, I recommend that you visit the GitHub related page at

<https://guides.github.com/introduction/flow/index.html>.



Other workflows

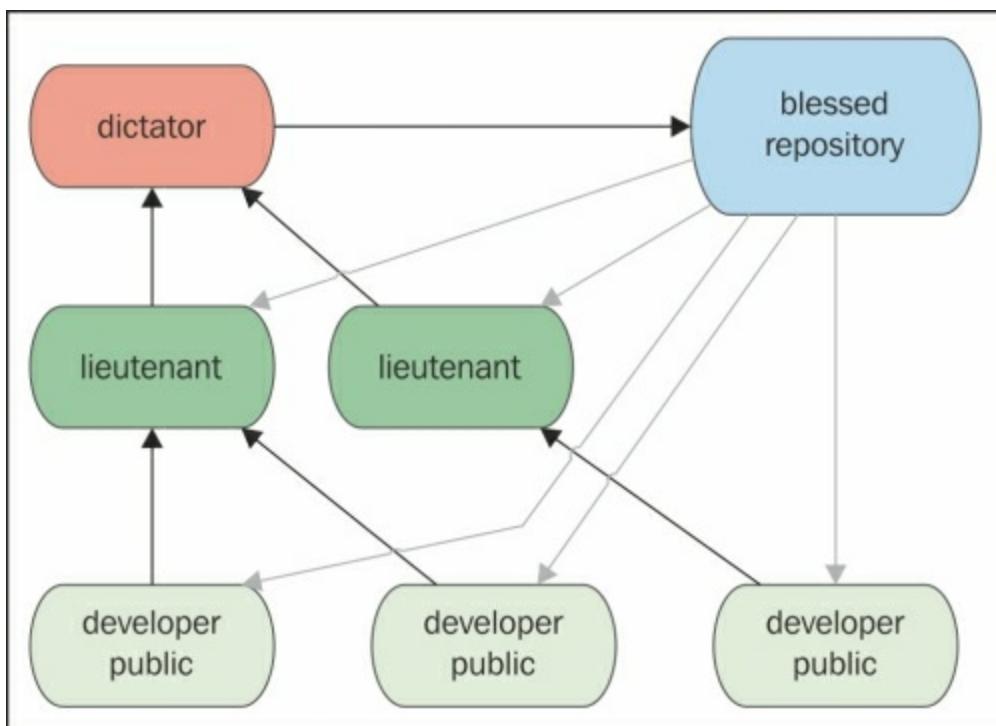
Obviously, there are many other workflows. I will spend just few words on the one that (fortunately) convinced *Linus Torvalds* to realize the Git VCS.

The Linux kernel workflow

The Linux kernel uses a workflow that refers to the traditional way in which *Linus Torvalds* has driven its evolution during these years. It is based on a military-like hierarchy.

Simple kernel developers work on their personal branches, rebasing the master branch in the reference repository. Then they push their branches to the lieutenant developer's master branch. Lieutenants are developers

who *Linus* assigned to particular topics and areas of the kernel because of their experience. When a lieutenants have done their work, they push it to the benevolent dictator master branch (**Linus branch**). Then, if things are OK (it is not simple to cheat him), *Linus* would push his master branch to the blessed repository, the one that developers use to rebase from, before starting their work.



Summary

In this chapter, we became aware of the effective ways to use Git. I personally consider this chapter the most important for a new Git user, because it applies some rules and discipline so that you will obtain the most from this tool. So, pick up a good workflow (make your own, if necessary), and pay attention to your commits. This is the only way to become a good versioning system user, not only in Git.

In the next chapter, we will see some tips and tricks to use Git even if you have to deal with Subversion servers. Then, we will take a quick look at migrating from Subversion to Git.

Chapter 6. Migrating to Git

In this chapter, we will try to migrate a Subversion repository into a Git one, preserving the changes history. Git and Subversion can coexist as Git has some dedicated commands to exchange data with Subversion, and you can even continue to use both.

The purpose of this chapter is to help developers who actually use Subversion to start using Git instantly, even if the rest of the team continues to use Subversion. In addition, the chapter covers definitive migration for people who decide to abandon Subversion in favor of Git.

Before starting

In the first part of this chapter, we will take a look at some good practices to keep safety and work on actual SVN repository with no hassles. Bear in mind that the purpose of this chapter is only to give readers some hints; dealing with big and complex repositories deserves a more prudent and articulated approach.

Prerequisites

To be able to do these experiments, you need a Subversion tool; on Windows, the most used tool is the well-known **TortoiseSVN** (available at <http://tortoisessvn.net>), which provides both command-line tools: GUI and shell integration.

I recommend to do a full installation of TortoiseSVN, including command-line tools as we'll need some of them to make experiments.

Working on a Subversion repository using Git

In the first part of this chapter, we will see the most cautious approach while starting to move away from Subversion, which is to keep the original repository using Git to fetch and push changes. For the purpose of learning, we will create a local Subversion repository, using both Subversion and Git to access its contents.

Creating a local Subversion repository

Without the hassle of remote servers, let's create a local Subversion repository as a container for our experiments:

```
$ cd \Repos  
$ svnadmin create MySvnRepo
```

Now there's nothing more and nothing less to be done here, and the repository is now ready to be filled with folders and files.

Checking out the Subversion repository with svn client

At this point, we have a working Subversion repository; we can now check it out in a folder of our choice, which will become our working copy; in my case, I will use the C:\Sources folder:

```
$ cd \Sources\svn  
$ svn checkout file:///Repos/MySvnRepo
```

You now have a MySvnRepo folder under the Sources folder, ready to be filled with your project files; but first let me remind you of a couple of things.

As you may know, a Subversion repository generally has the following subfolder structure:

- /trunk: This is the main folder, where generally you have the code under development
- /tags: This is the root folder of the snapshots you usually freeze and leave untouched, for example /tags/v1.0
- /branches: This is the root folder of all the repository branches you will create for feature development, for example
/branches/NewDesign

Subversion does not provide a command to initialize a repository with this layout (commonly known as **standard layout**), so we have to build it up by hand.

At this point, we can import a skeleton folder that already contains the three subfolders (/trunk, /branches, and /tags) with a command like this:

```
$ cd \Sources\svn\MySvnRepo
$ svn import /path/to/some/skeleton/dir
```

Otherwise, we can create folders by hand using the `svn mkdir` command:

```
$ cd \Sources\svn\MySvnRepo
$ svn mkdir trunk
$ svn mkdir tags
$ svn mkdir branches
```

Commit the folders we just created and the repository is ready:

```
svn commit -m "Initial layout"
```

Now add and commit the first file, as shown in the following code:

```
$ cd trunk
$ echo "This is a Subversion repo" > readme.txt
$ svn add readme.txt
$ svn commit -m "Readme file"
```

Feel free to add more files or import an existing project if you want to replicate a more real situation; for import files in a Subversion repository, you can use the `svn import` command, as you saw before:

```
$ svn import \MyProject\Folder
```

Later, we will add a tag and a branch to verify how Git interacts with them.

Cloning a Subversion repository from Git

Git provides a set of tools to cooperate with Subversion; the base command is actually `git svn`; with `git svn` you can clone Subversion repositories, retrieve and upload changes, and much more.

So, wear the Git hat and clone the Subversion repository using the `git svn clone` command:

```
$ cd \Sources\git
$ git svn clone file:///Repos/MySvnRepo
```

`git svn clone` usually runs smoothly on Linux boxes, while in Windows you get a weird error message, as shown here:

```
Couldn't open a repository: Unable to open an ra_local session
to URL
```

This happens because in Windows there are some problems in the `git svn` command backport while using the `file://` protocol.

Setting up a local Subversion server

To bypass this problem, we can use the `svn://` protocol instead of `file://`.

To do this, we will use the `svnserve` command to start a Subversion server exposing our repositories directory root. But first we have to edit some files to set up the server.

The main file is the `svnserve.conf` file; look at your `MySvnRepo` repository folder, `C:\Repos\MySvnRepo` in this case, jump into the `conf` subfolder, and edit the file uncommenting these lines (remove the starting #):

```
anon-access = read
auth-access = write
password-db = C:\Repos\MySvnRepo\passwd
authz-db = C:\Repos\MySvnRepo\authz
realm = MySvnRepo
```

Normally, full paths are unnecessary for the `passwd` and `authz` files, but in Windows I find them mandatory, otherwise the Subversion server does not load them.

Now edit the `passwd` file, which contains the user's credentials, and add a new user:

```
[users]
adminUser = adminPassword
```

Then edit the `authz` file and set up groups and rights for the user:

```
[groups]
Admins = admin
```

Continuing with the `authz` file, set rights for repositories; the Admin will have read/write access to all repositories (the section `[/]` means "all repository"):

```
[/]
@Admins = rw
* =
```

At this point, we can start the server with this command:

```
$ svnserve -d --config-file
C:\Repos\MySvnRepo\conf\svnserve.conf --root C:\Repos
```

`-d` starts the server as a daemon; `--config-file` specifies the config file to load and `--root` tells the server where the main root folder of all your repositories is.

Now we have a Subversion server responding at `svn://localhost:3690`. We can finally clone the repository using Git:

```
$ cd \Sources\git
```

```
$ git svn clone svn://localhost/MySvnRepo/trunk
```

This time things will go smoothly; we are now talking with a Subversion server using Git.

Adding a tag and a branch

Just to have a more realistic situation, I will add a tag and a branch; in this manner, we will see how to deal with them in Git.

So, add a new file:

```
$ echo "This is the first file" > svnFile01.txt
$ svn add svnFile01.txt
$ svn commit -m "Add first file"
```

Then tag this snapshot of the repository as v1.0 as you know that in Subversion, a tag or a branch is a copy of a snapshot:

```
$ echo "This is the first file" > svnFile01.txt
$ svn add svnFile01.txt
$ svn copy svn://localhost/MySvnRepo
svn://localhost/MySvnRepo/tags/v1.0 -m "Release 1.0"
```

Committing a file to Subversion using Git as a client

Now that we have a running clone of the original Subversion repository, we can use Git as it was a Subversion client. So add a new file and commit it using Git:

```
$ echo "This file comes from Git" >> gitFile01.txt
$ git add gitFile01.txt
$ git commit -m "Add a file using Git"
```

Now we have to push this file to Subversion:

```
$ git svn dcommit
```

Well done! We can even use Git to fetch changes with the `git svn fetch` command, update the local code using the `git svn rebase`

command, and so on; for other commands and options, I recommend that you read the main page of `git svn --help`.

Using Git as a Subversion client is not the best we can obtain, but at least it is a way to start using Git even if you cannot abandon Subversion instantly.

Using Git with a Subversion repository

Using Git as a client of Subversion can raise some confusion due to the flexibility of Git as compared to the more rigid way Subversion organizes files. To be sure to maintain a Subversion-friendly way of work, I recommend that you follow some simple rules.

First of all, be sure your Git `master` branch is related to the `trunk` branch in Subversion; as we already said, Subversion users usually organize a repository in this way:

- a `/trunk` folder, which is the main folder
- a `/branches` root folder, where you put all the branches, each one located in a separate subfolder (for example, `/branches/feat-branch`)
- a `/tags` root folder, where you collect all the tags you made (for example, `/tags/v1.0.0`)

To adhere to this layout, you can use the `--stdlayout` option when you're cloning a Subversion repository:

```
$ git svn clone <url> --stdlayout
```

In this manner, Git will hook the `/trunk` Subversion branch to the Git `master` branch, replicating all the `/branches` and `/tags` branches in your local Git repository and allowing you to work with them in a 1:1 synchronized context.

Migrating a Subversion repository

When possible, it is recommended to completely migrate a Subversion repository to Git; this is quite simple to do and mostly depends on the size of the Subversion repository and the organization.

If the repository follows the standard layout as described before, a migration is only a matter of minutes.

Retrieving the list of Subversion users

If your Subversion repository has been used from different people, you are probably interested in preserving the commit author's name, which is true even in the new Git repository.

If you have the `awk` command available (maybe using Cygwin in Windows), there is a script here that fetches all the users from Subversion logs and appends them to a text file we can use in Git while cloning to perfectly match the Subversion users, even in Git-converted commits:

```
$ svn log -q | awk -F '||' '/^r/ {sub("^ ", "", $2); sub(" $", "", $2); print $2" = \"$2\" <\"$2\">"}' | sort -u > authors.txt
```

Now we will use the `authors.txt` file in the next cloning step.

Cloning the Subversion repository

To begin the migration, we have to locally clone the Subversion repository as we did before; I recommend once more adding the `--stdlayout` option to preserve the branches and tags and then to add the `-A` option to let Git convert commit authors while cloning:

```
$ git svn clone <repo-url> --stdlayout --prefix svn/ -A  
authors.txt
```

In case the Subversion repository has trunks, branches, and tags located in other paths (with no standard layout), Git provides you with a way to specify them with the `--trunk`, `--branches`, and `--tags` options:

```
$ git svn clone <repo-url> --trunk=<trunk-folder> --branches=<branches-subfolder> --tags=<tags-subfolder>
```

When you fire the `clone` command, remember that this operation can be time-consuming; in a repository with 1000 commits, it is not unusual to wait 15 to 30 minutes for this.

Preserving the ignored file list

To preserve the previously ignored files in Subversion, we can append the `svn:ignore` settings to the `.gitignore` file:

```
$ git svn show-ignore >> .gitignore
$ git add .gitignore
$ git commit -m "Convert svn:ignore properties to .gitignore"
```

Pushing to a local bare Git repository

Now that we have a local copy of our repository, we can move it to a brand new Git repository. Here you can already use a remote repository on your server of choice, which can even be GitHub or Bitbucket, but I recommend that you use a local bare repository; we may as well do some other little adjustments (like renaming tags and branches) before pushing files to a blessed repository. So, first initialize a bare repository in a folder of your choice:

```
$ mkdir \Repos\MyGitRepo.git
$ cd \Repos\MyGitRepo.git
$ git init --bare
```

Now make the default branch to match the Subversion `trunk` branch name:

```
$ git symbolic-ref HEAD refs/heads/trunk
```

Then add a `bare` remote pointing to the bare repository just created:

```
$ cd \Sources\MySvnRepo  
$ git remote add bare file:///C/Repos/MyGitRepo.git
```

Finally push the local cloned repository to the new bare one:

```
$ git push --all bare
```

We now have a brand new bare repository that is a perfect copy of the original Subversion repository. We can now adjust branches and tags to better fit the usual Git layout.

Arranging branches and tags

Now we can rename branches and tags to obtain a more Git-friendly scenario.

Renaming the trunk branch to master

The Subversion main development branch is `/trunk`, but in Git, as you know, we prefer to call the main branch `master`; here's a way to rename it:

```
$ git branch -m trunk master
```

Converting Subversion tags to Git tags

Subversion treats tags as branches; they are all copies of a certain trunk snapshot. In Git, on the contrary, branches and tags have a different significance.

To convert Subversion tags and branches into Git tags, the following simple script does the work:

```
$ git for-each-ref --format='%(refname)' refs/heads/tags |  
cut -d / -f 4 |  
while read ref  
do  
  git tag "$ref" "refs/heads/tags/$ref";  
  git branch -D "tags/$ref";  
done
```

Pushing the local repository to a remote

You now have a local bare Git repository ready to be pushed to a remote server; the result of the conversion is a full Git repository, where branches, tags, and commit history have been preserved. The only thing you have to do by hand is to eventually accommodate Git users.

Comparing Git and Subversion commands

Here you can find a short and partial recap table, where I try to pair the most common Subversion and Git commands to help Subversion users to quickly shift their minds from Subversion to Git.

Subversion	Git
Creating a repository	
<code>svnadmin create <repo-name></code> <code>svnadmin import <project-folder></code>	<code>git init <repo-name></code> <code>git add .</code> <code>git commit -m "Initial commit"</code>
Getting the whole repository for the first time	
<code>svn checkout <url></code>	<code>git clone <url></code>
Inspecting local changes	
<code>svn status</code> <code>svn diff less</code>	<code>git status</code> <code>git diff</code>
Dealing with files (adding, removing, moving)	
<code>svn add <file></code> <code>svn rm <file></code> <code>svn mv <file></code>	<code>git add <file></code> <code>git rm <file></code> <code>git mv <file></code>
Committing local changes	
<code>svn commit -m "<message>"</code>	<code>git commit -a -m "<message>"</code>
Reviewing history	
<code>svn log less</code>	<code>git log</code>

```
svn blame <file>
```

```
git blame <file>
```

Branching and Tagging

```
svn copy <source> <branch-name> svn copy <source> <tag-name>
```

```
git branch <branch-name>  
git tag -a <tag-name>
```

Remember: in Subversion tags and branches represent physical copies of a source branch (the trunk, another branch or another tag), while in Git a tag is only a pointer to a particular commit.

Merging

(assuming the branch was created in revision 42 and you are inside a working copy of trunk)

```
svn merge -r 42:HEAD <branch>
```

```
git merge <branch>
```

Summary

This chapter barely scratches the surface, but I think it'll be useful to get a sense of the topic. If you have wide Subversion repositories, you will probably need better training before starting to convert them to Git, but for small to medium ones, now you know the fundamentals to begin with.

The only suggestion I want to share with you is to not be in a hurry; start by letting Git cooperate with your Subversion server, reorganize your repository when it's messy, do a lot of backups, and finally try to convert it; you will convert it more than once, as I did, but in the end you will get more satisfaction from doing that.

In the next chapter, I will share with you some useful resources I found during my career as a Git user.

Chapter 7. Git Resources

This chapter is a collection of resources I built during my experience with Git. I will share some thoughts about GUI tools, web interfaces with Git repositories, and learning resources, hoping they will act as a springboard for a successful Git career.

Git GUI clients

When beginning to learn a new tool, especially a wide and complex one like Git, it can be useful to take advantage of some GUI tools that are able to picture commands and patterns in a way that is more simple to understand.

Git benefits from a wide range of GUI tools, so it's only a matter of choice. I want to tell you right away that there is no perfect tool, as frequently happens, but there are enough of them to pick the one that fits your needs better.

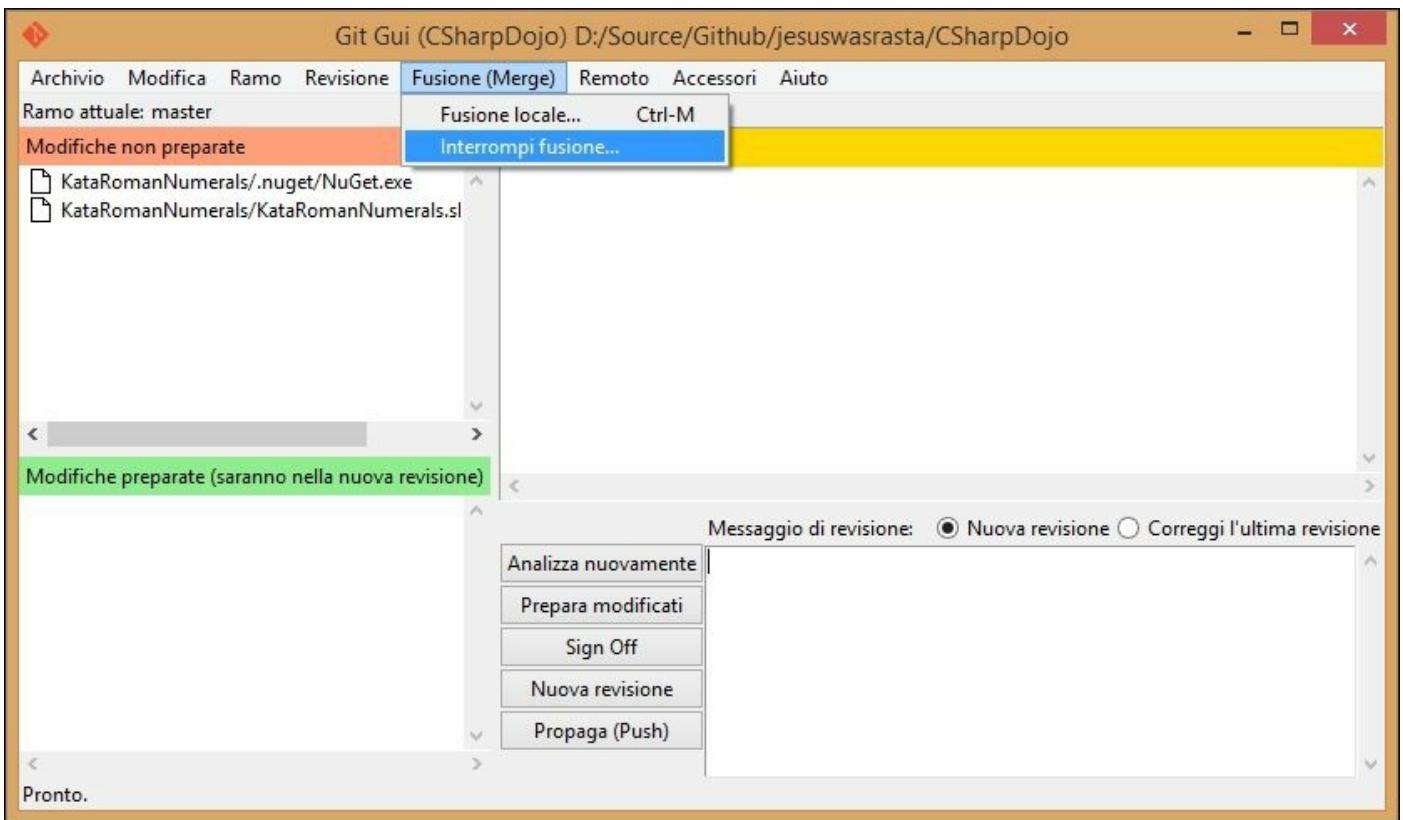
Windows

As a Microsoft .NET developer, I use Windows 99 percent of the time. In spare time, I play a little bit with Linux, but in that case I prefer to use the command line. In this section, you will find tools I use or I have used in the past, while in the other platform section I will provide only some hints based on words of other people.

Git GUI

Git has an integrated GUI, as we learnt from the previous chapters. Probably, it is not one of the most eye-catching solution you will find, but for small issues it can be enough. The reason for using it is that it is already installed when you install Git, and that it is well integrated even with the command prompt; so for blaming files, see history or interactive merging can be fired easily (just type `git gui <command>` on your shell). But I have to come clean: I don't like it much.

In Windows, Git GUI will be installed following the language or the region you specified on Windows. What's the problem? Well, the problem is that in non-English languages, they translate everything - even command names! In Italian, instead of merge I see **Fusione**; pushing gets translated as **Propaga**; and untracked files become **Modifiche non preparate**. The problem is not the translation itself, but simply that I find translating concepts or command names (like even Windows and other Microsoft tools did in the past) perplexing; it is confusing and counterproductive.

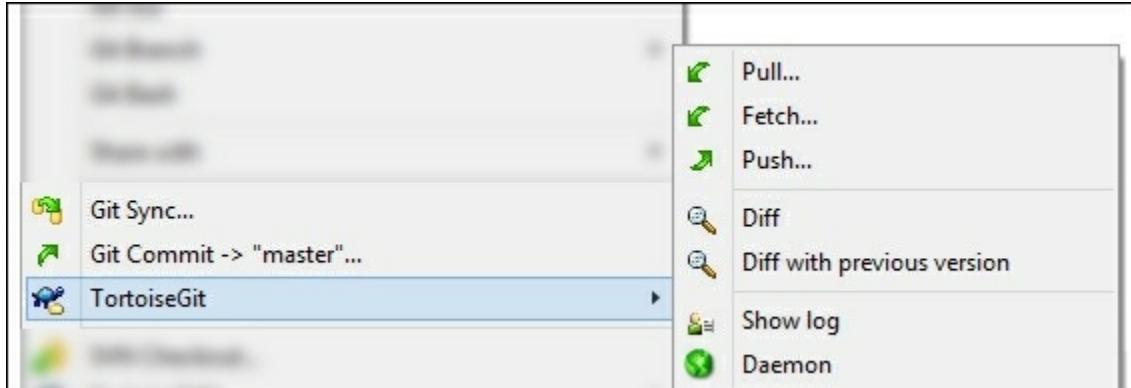


TortoiseGit

If you have migrated from using Subversion to using Git, you have probably already heard about **TortoiseSVN**, a well-crafted tool for dealing with Subversion commands directly from Explorer, through the right-click shell integration.

TortoiseGit brings Git, instead of Subversion, to the same place. By installing TortoiseGit, you will benefit from the same Explorer

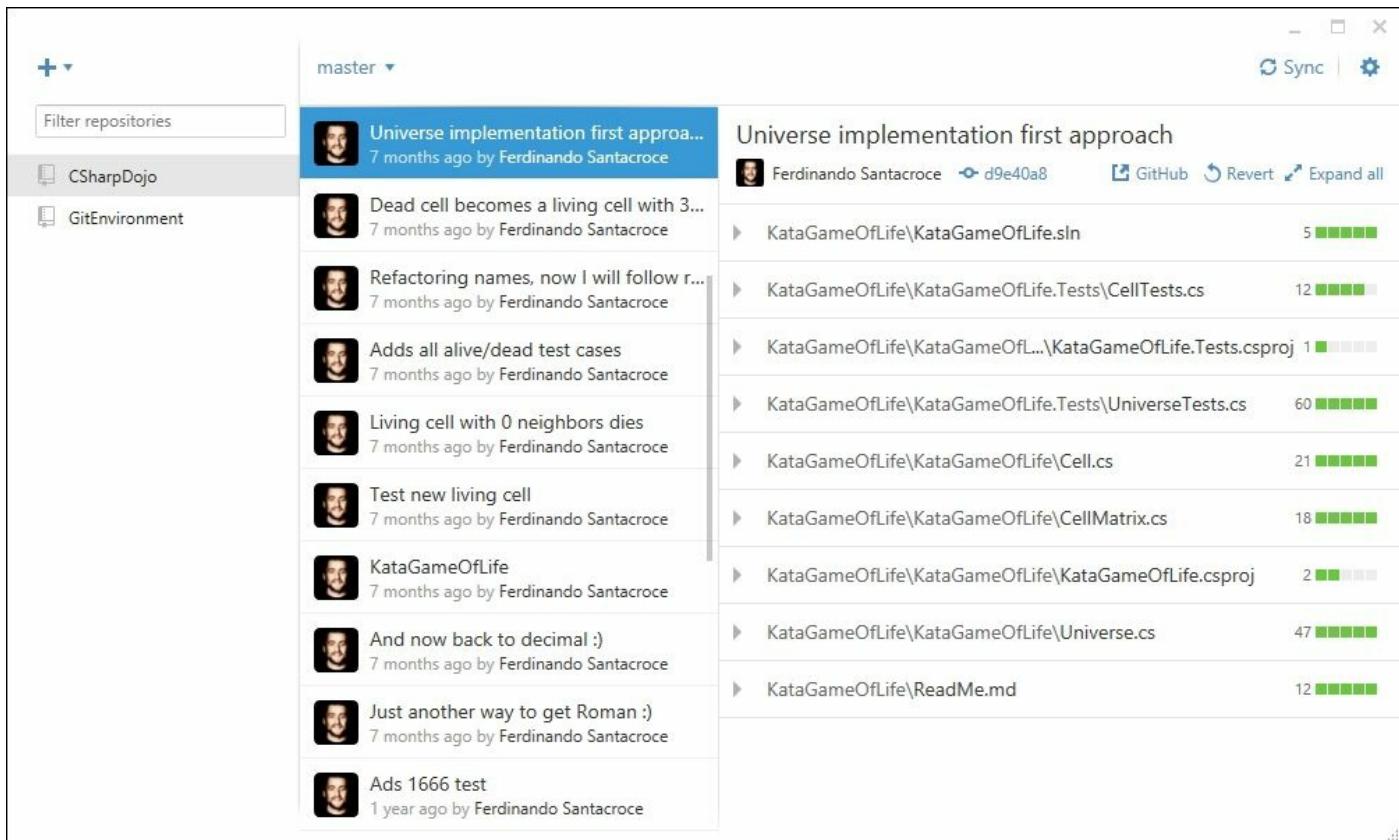
integration, leaving most Git commands only a step away from you. Even if I discourage you to use localized versions, TortoiseGit is available in different languages. Also, bear in mind that you need to install Git in advance as it is not included in the TortoiseGit setup package.



GitHub for Windows

GitHub offers a stylish Modern UI based client. I have to admit that I snubbed it at first, mostly because I was sure that I could use it only for GitHub repositories. However, I realized that one can use it even with other remotes, but it's clear that the client is tailored for GitHub. To use other remotes, you have to edit the config file by hand, substituting the GitHub remote with the one you want.

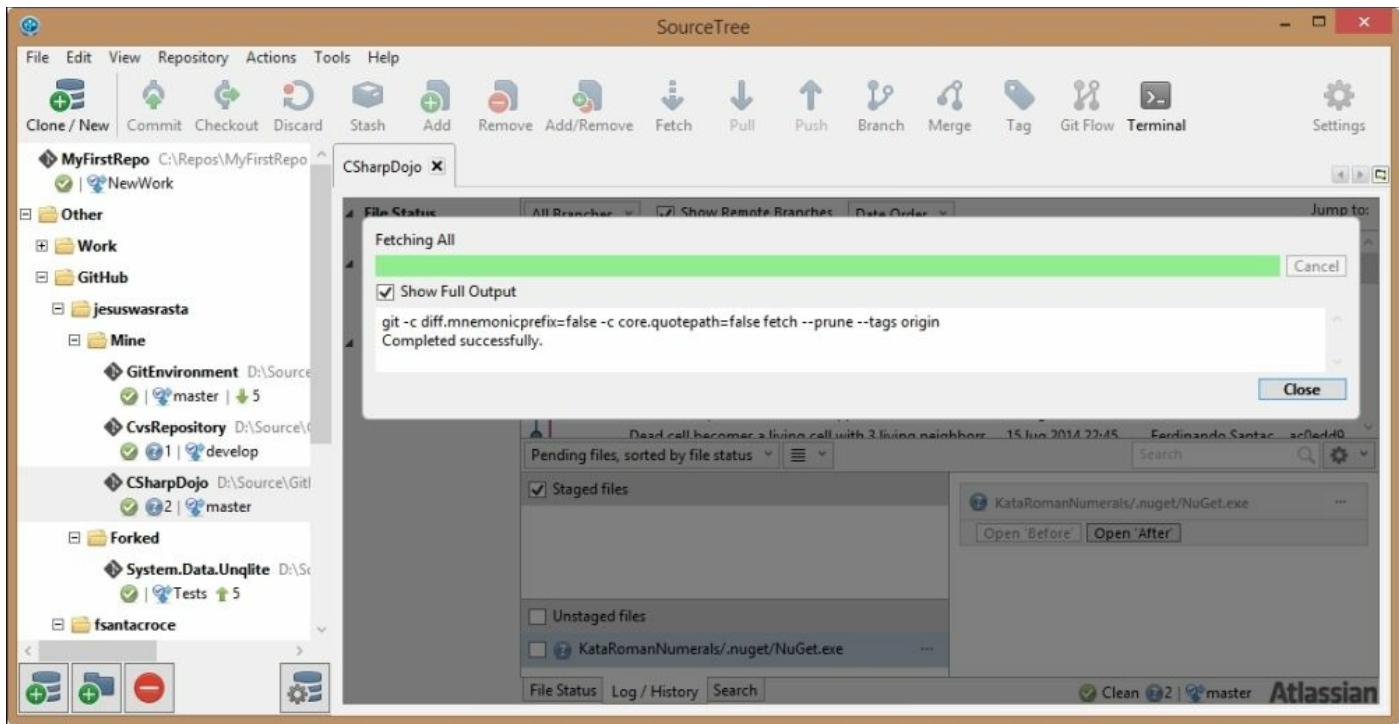
If you want a general purpose client, probably this is not the best tool for you; but if you work mostly on GitHub, it may likely be the best GUI in the market.



Atlassian SourceTree

This is my favorite client. SourceTree is free like all the other tools; it comes from the mind of Atlassian, the well-known company behind Bitbucket and other popular services like Jira, Confluence, and Stash. SourceTree can handle all kinds of remotes, offering facilities (like remembering passwords) to access the most popular services like Bitbucket and GitHub.

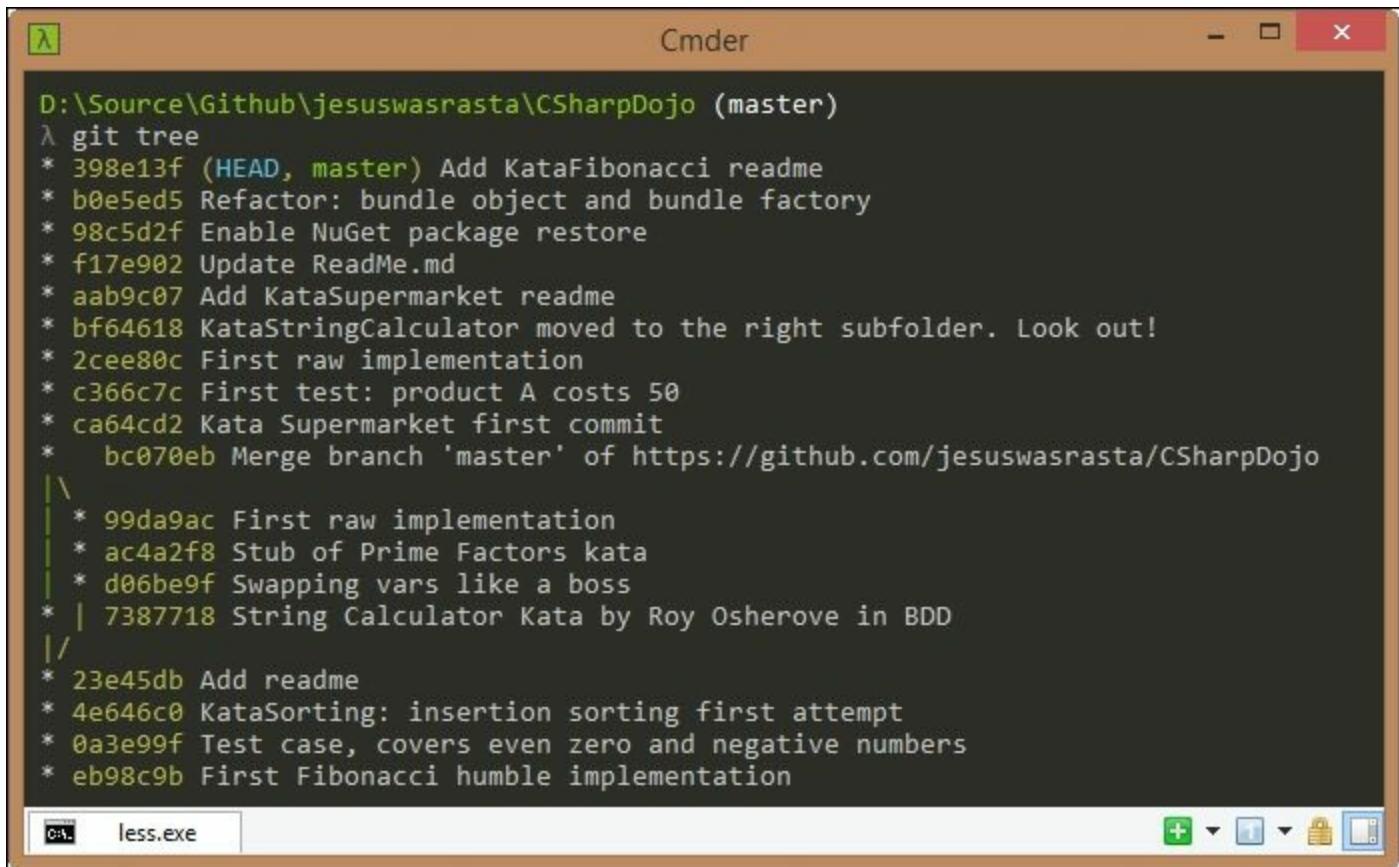
It embeds the GitFlow way of organizing repositories by design, offering a convenient button to initialize a repository with GitFlow branches, and integrating GitFlow commands provided by the author. The most interesting thing I found at first was that you can enable a window where SourceTree shows the equivalent Git command when you use some of Git commands by user interface; in this manner, when you doubt you can remember the right command for the job, you can use SourceTree to accomplish your task and see what commands it uses to get the work done.



SourceTree is available even for Mac OS X.

Cmder

Cmder is not really a Git GUI, but a nicer portable console emulator you can use instead of the classic Bash shell:

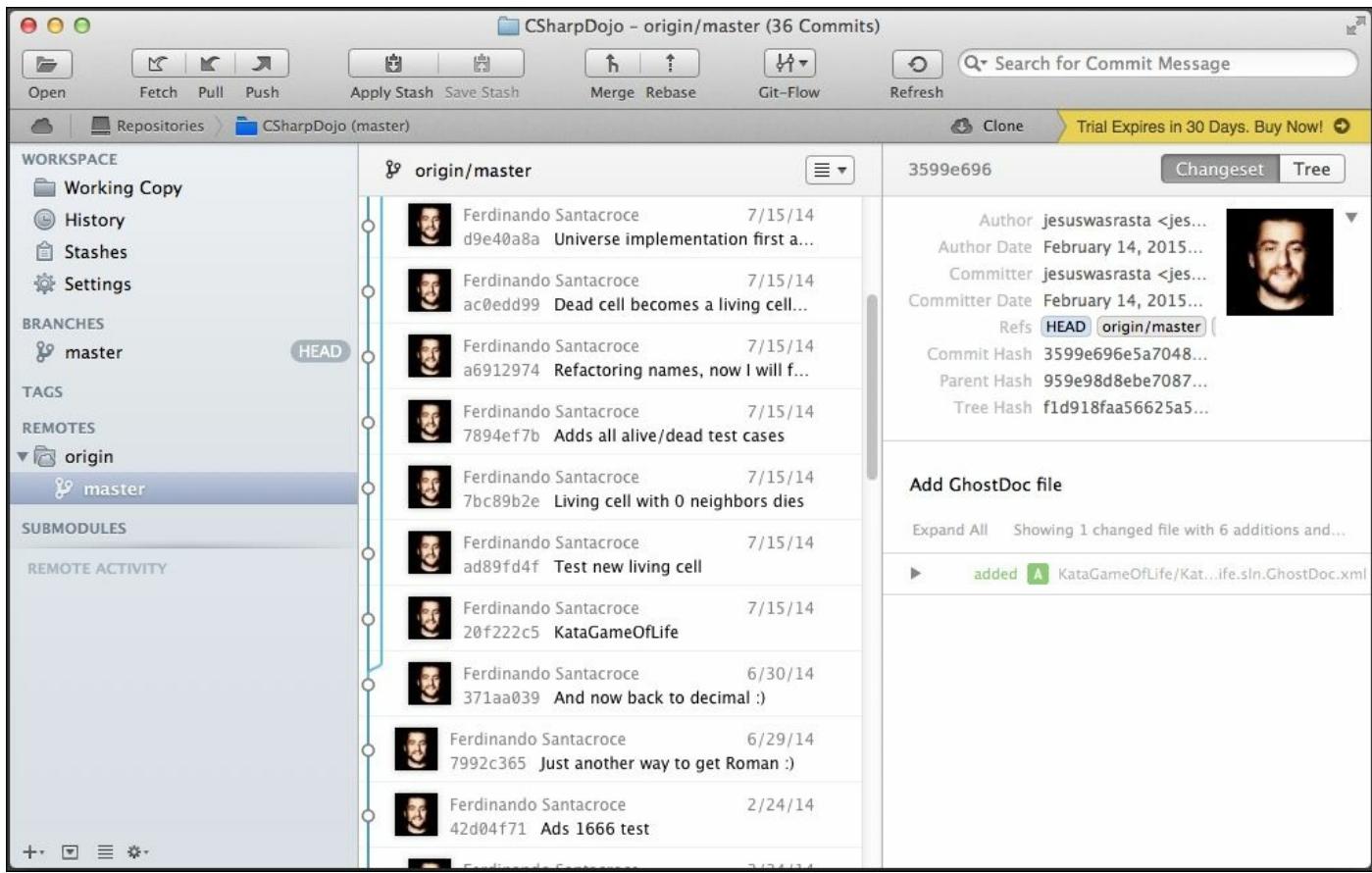


D:\Source\Github\jesuswasrasta\CSharpDojo (master)
λ git tree
* 398e13f (HEAD, master) Add KataFibonacci readme
* b0e5ed5 Refactor: bundle object and bundle factory
* 98c5d2f Enable NuGet package restore
* f17e902 Update ReadMe.md
* aab9c07 Add KataSupermarket readme
* bf64618 KataStringCalculator moved to the right subfolder. Look out!
* 2cee80c First raw implementation
* c366c7c First test: product A costs 50
* ca64cd2 Kata Supermarket first commit
* bc070eb Merge branch 'master' of https://github.com/jesuswasrasta/CSharpDojo
|
| * 99da9ac First raw implementation
| * ac4a2f8 Stub of Prime Factors kata
| * d06be9f Swapping vars like a boss
* | 7387718 String Calculator Kata by Roy Osherove in BDD
|/
* 23e45db Add readme
* 4e646c0 KataSorting: insertion sorting first attempt
* 0a3e99f Test case, covers even zero and negative numbers
* eb98c9b First Fibonacci humble implementation

It looks nicer than the original shell; it has multi-tab support and a wide set of configuration options to let you customize it as you prefer, thanks to ConEmu and Clink projects. Finally yet importantly, it comes with Git embedded. You can download it from GitHub at <https://github.com/bliker/cmder>.

Mac OS X

As I already said, I have no experience with Mac OS X Git clients; the only information I can share with you is that GitHub offers its client for free, even for Mac, like it does Atlassian with SourceTree. There is no TortoiseGit for Mac, but I have heard about a cool app called Git Tower. Please consider giving it a try as it seems very well crafted.



Another great tool is SmartGit, available for free for open-source projects: <http://www.syntevo.com/smartygit/>

Linux

Linux is the reason for Git, so I think that it is the best place to work with Git. I play with Linux now and then, and I usually use the Bash shell for Git.

For the ZSH shell lover, I suggest looking at <http://ohmyz.sh/>, an interesting open source project where you can find tons of plugins and themes. About plugins, there are some of them that let you enhance your Git experience with this famous alternate console.

You can take a look at some Git GUI for Linux at

<http://git-scm.com/download/gui/linux>

Building up a personal Git server with web interface

In the office where I work, I was the first person who started to use Git for production code. At some point, after months of little trials in my spare time, I gained courage and converted all the Subversion repositories, where I usually work alone, into Git ones.

Unfortunately, firm IT policies forbid me to use external source code repositories; so no GitHub or Bitbucket. To make things even worse, I also could not obtain a Linux server, and take advantage of great web interfaces like Gitosis, GitLab, and so on. So I started to Google around the web for a solution, and I finally found a solution that can be useful for people in a similar condition.

The SCM Manager

SCM Manager (<https://www.scm-manager.org/>) is a very easy solution to share your Git repositories in a local Windows network. It offers a standalone solution to install and make it work on top of Apache Web Server directly in Windows. Though it is built in Java, you can make it work even in Linux or Mac.

It can manage Subversion, Git, and Mercurial repositories, allowing you to define users, groups, and so on. It has a good list of plugins too for other version control systems and other development related tools like Jenkins, Bamboo, and so on. There's also a Gravatar plugin and an Active Directory plugin, to let you and other colleagues use default domain credentials to access your internal repositories.

I've been using this solution for about two years without a hitch, excluding only some configuration related annoyances during updates, due to my custom path personalization.

SCM Manager

Navigation

- Main**
 - Repositories
 - Import Repositories
- Config**
 - General
 - Repository Types
 - Plugins
- Security**
 - Change Password
 - Users
 - Groups
- Log out**
 - Log out

Repositories

Add Remove Reload Filter: Search:

Name	Type	Contact	Description	Creation date	Url
79	Mercurial			2011-12-15 21:09:30	http://hades.uasw.edu:8081/scm/hg/Issues/79
83	Subversion			2011-12-28 18:09:32	http://hades.uasw.edu:8081/scm/svn/Issues...

issues (2 Repositories)

scm-git	Git	s.sdorra@gmail.com	SCM-Manager Git ...	2011-05-06 12:56:47	http://hades.uasw.edu:8081/scm/git/scm/sc...
scm-graph-plugin	Mercurial			2011-09-27 22:46:37	http://hades.uasw.edu:8081/scm/hg/scm/sc...
scm-gravatar-plugin	Mercurial	s.sdorra@gmail.com	SCM-Manager Gra...	2011-07-06 08:24:05	http://hades.uasw.edu:8081/scm/hg/scm/sc...
scm-ldap-plugin	Mercurial			2011-09-28 08:13:34	http://hades.uasw.edu:8081/scm/hg/scm/sc...
scm-manager	Mercurial	s.sdorra@gmail.com	SCM-Manager	2011-05-06 12:55:59	http://hades.uasw.edu:8081/scm/hg/scm/sc...
scm-pam-plugin	Mercurial	s.sdorra@gmail.com	SCM-Manager PAM...	2011-07-06 08:21:05	http://hades.uasw.edu:8081/scm/hg/scm/sc...
scm-scala-test	Mercurial	s.sdorra@gmail.com	Creating SCM-Man...	2011-07-06 08:19:56	http://hades.uasw.edu:8081/scm/hg/scm/sc...

scm (7 Repositories)

external	Subversion			2011-12-01 20:34:43	http://hades.uasw.edu:8081/scm/svn/sub/ex...
git-mod-1	Git			2012-01-07 14:23:59	http://hades.uasw.edu:8081/scm/git/sub/git-...
main	Mercurial			2011-12-10 14:34:28	http://hades.uasw.edu:8081/scm/hg/sub/main
main-git	Git			2012-01-07 14:19:20	http://hades.uasw.edu:8081/scm/git/sub/main
module-1	Mercurial			2011-12-10 14:34:41	http://hades.uasw.edu:8081/scm/hg/sub/module-1

sub (5 Repositories)

external	Subversion			2011-12-01 20:34:43	http://hades.uasw.edu:8081/scm/svn/sub/ex...
git-mod-1	Git			2012-01-07 14:23:59	http://hades.uasw.edu:8081/scm/git/sub/git-...
main	Mercurial			2011-12-10 14:34:28	http://hades.uasw.edu:8081/scm/hg/sub/main
main-git	Git			2012-01-07 14:19:20	http://hades.uasw.edu:8081/scm/git/sub/main
module-1	Mercurial			2011-12-10 14:34:41	http://hades.uasw.edu:8081/scm/hg/sub/module-1

scm/scm-manager Settings Permissions Sub Repositories

Name: scm/scm-manager
Type: Mercurial (hg)
Contact: s.sdorra@gmail.com
Url: <http://hades.uasw.edu:8081/scm/hg/scm/scm-manager>
Checkout: hg clone <http://scmadmin@hades.uasw.edu:8081/scm/hg/scm/scm-manager>

Commits, Source

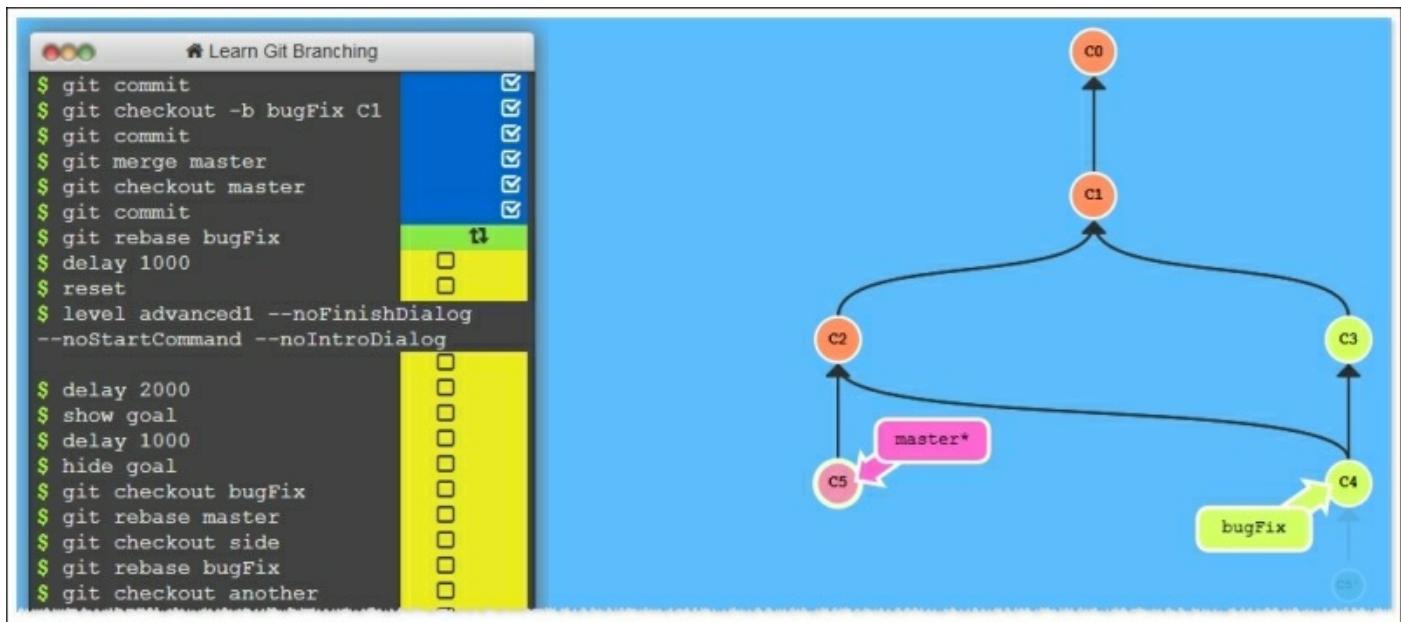
© SCM Manager

1.12-SNAPSHOT

Learning Git in a visual manner

The last thing I would like to share with readers is a web app I found useful at the very beginning, to better understand the way Git works.

LearnGitBranching (<http://pcottle.github.io/learnGitBranching/>) is a tremendously helpful web app that offers you some exercises to help you grow your Git culture. Starting from basic commit exercises, you learn how to branch, rebase, and so on. However, the really cool thing is that at the right side of the page, you will see a funny repository graph evolving in real time, following the commands you type in the emulated shell.



Git on the Internet

In the end, I would suggest you to follow some resources that I usually follow to learn new things and get in touch with other smart and funny Git users over the Internet.

Git community on Google+

This community is full of people who are happy to share their knowledge with you; most of the coolest things I know about Git have been discovered here, at:

<https://plus.google.com/u/0/communities/112688280189071733518>

GitMinutes and Thomas Ferris Nicolaisen's blog

Thomas is a skilled Git user, and a very kind person. On his blog, you will find many interesting resources, including videos where he talks about Git at local German programming events.

More than this, Thomas runs GitMinutes podcast series, where he talks about Git with other people, discussing general purpose topics, tools opinions, and so on.

Take a look at www.tfnico.com and www.gitminutes.com.

Ferdinando Santacroce's blog

On my personal blog, <http://jesuswasrasta.com/>, I recently started a *Git Pills* series, where I share with readers some things I discovered using Git, quick techniques to get the job done, and how to recover from weird situations.

Summary

In this chapter, we went through some Git GUI clients. Even if I encourage people to understand Git by using shell commands, I have to admit that for most common tasks, using a GUI based tool makes me feel more comfortable, especially when diffing or reviewing history.

Then we discovered that we could obtain a personal Git server with a fancy web interface: the Internet has plenty of good pieces of software to achieve this target.

At the end, like my last suggestion, I mentioned some good resources to enhance your Git comprehension: all the knowledge fields, hearing the voice of the experts, and asking them questions is the most effective way to get your work done.

Part II. Module 2

Git Version Control Cookbook

90 hands-on recipes that will increase your productivity when using Git as a version control system

Chapter 1. Navigating Git

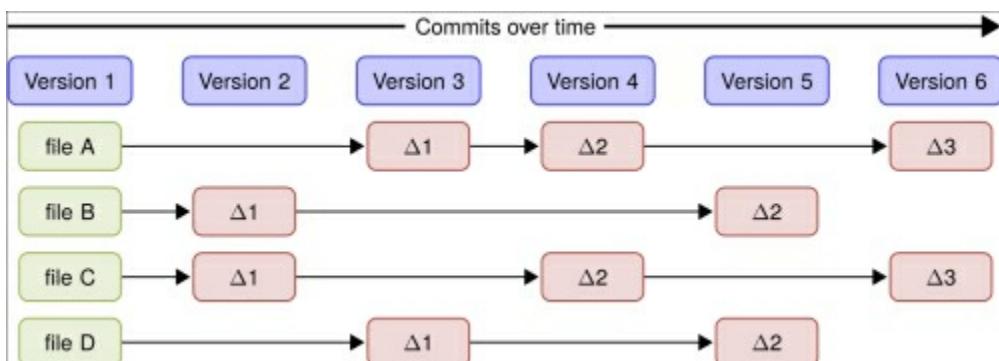
In this chapter, we will cover the following topics:

- Git's objects
- The three stages
- Viewing the DAG
- Extracting fixed issues
- Getting a list of the changed files
- Viewing the history with Gitk
- Finding commits in the history
- Searching through the history code

Introduction

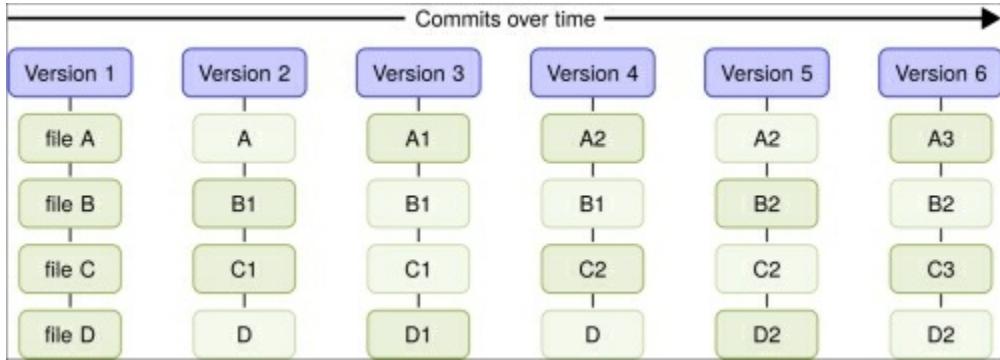
In this chapter, we will take a look at Git's data model. We will learn how Git references its objects and how the history is recorded. We will learn how to navigate the history, from finding certain text snippets in commit messages to the introduction of a certain string in the code.

The data model of Git is different from other common **version control systems (VCSs)** in the way Git handles its data. Traditionally, a VCS will store its data as an initial file followed by a list of patches for each new version of the file.



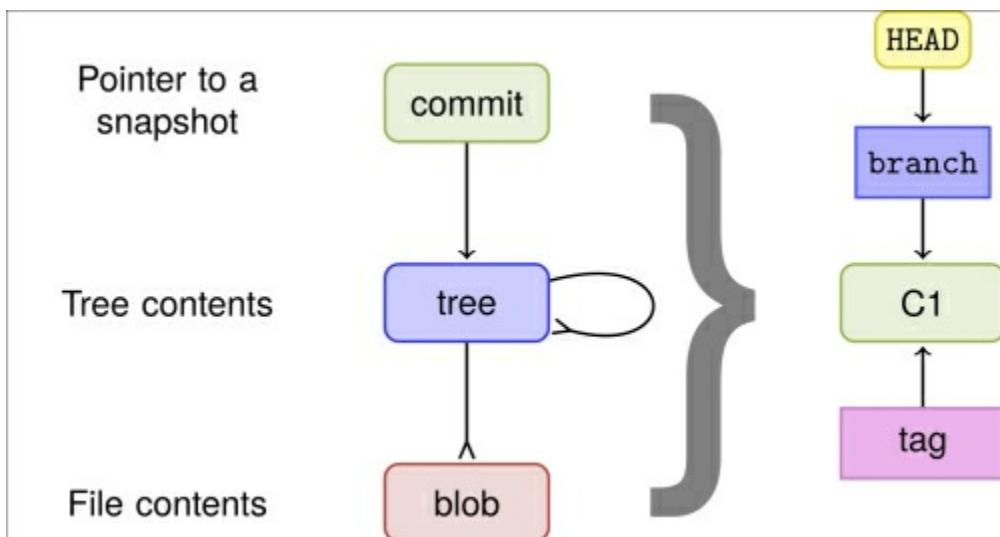
Git is different; instead of the regular file and patches list, Git records a snapshot of all the files tracked by Git and their paths relative to the

repository root, that is, the files tracked by Git in the file system tree. Each commit in Git records the full tree state. If a file does *not* change between commits, Git will not store the file once more; instead, Git stores a link to the file.



This is what makes Git different from most other VCSs, and in the following chapters, we will explore some of the benefits of this powerful model.

The way Git references the files and directories it tracks is directly built into the data model. In short, the Git data model can be summarized as shown in the following diagram:



The commit object points to the root tree. The root tree points to subtrees and files. Branches and tags point to a commit object and the

`HEAD` object points to the branch that is currently checked out. So for every commit, the full tree state and snapshot are identified by the root tree.

Git's objects

Now that you know Git stores every commit as a full tree state or snapshot, let's look closer at the object's Git store in the repository.

Git's object storage is a key-value storage, the key being the ID of the object and the value being the object itself. The key is an SHA-1 hash of the object, with some additional information such as size. There are four types of objects in Git, branches (which are not objects, but are important), and the special `HEAD` pointer that refers to the branch/commit currently checked out. The four object types are as follows:

- Files, or blobs as they are also called in the Git context
- Directories, or trees in the Git context
- Commits
- Tags

We will start by looking at the most recent commit object in the repository we just cloned, keeping in mind that the special `HEAD` pointer points to the branch currently checked out.

Getting ready

To view the objects in the Git database, we first need a repository to be examined. For this recipe, we will clone an example repository located here:

```
$ git clone https://github.com/dvaske/data-model.git  
$ cd data-model
```

Now you are ready to look at the objects in the database, we will start by looking first at the commit object, then the trees, the files, and finally the branches and tags.

How to do it...

Let's take a closer look at the object's Git stores in the repository.

The commit object

The special Git object `HEAD` always points to the current snapshot/commit, so we can use that as a target for our request of the commit we want to have a look at:

```
$ git cat-file -p HEAD
tree 34fa038544bcd9aed660c08320214bafff94150b
parent a90d1906337a6d75f1dc32da647931f932500d83
author Aske Olsson <aske.olsson@switch-gears.dk> 1386933960
+0100
committer Aske Olsson <aske.olsson@switch-gears.dk> 1386941455
+0100
```

This is the subject line of the commit message

It should be followed by a blank line then the body, which is this text. Here you can have multiple paragraphs etc. and explain your commit. It's like an email with subject and body, so get people's attention in the subject

The `cat-file` command with the `-p` option pretty prints the object given on the command line; in this case, `HEAD`, which points to `master`, which in turn points to the most-recent commit on the branch.

We can now see the commit object, consisting of the root tree (`tree`), the parent commit object's ID (`parent`), author and timestamp information (`author`), committer and timestamp information (`committer`), and the commit message.

The tree object

To see the tree object, we can run the same command on the tree, but with the tree ID (`34fa038544bcd9aed660c08320214bafff94150b`) as the target:

```
$ git cat-file -p 34fa038544bcd9aed660c08320214bafff94150b
100644 blob f21dc2804e888fee6014d7e5b1ceee533b222c15
README.md
040000 tree abc267d04fb803760b75be7e665d3d69eed32f8
a_sub_directory
100644 blob b50f80ac4d0a36780f9c0636f43472962154a11a
```

```

another-file.txt
100644 blob 92f046f17079aa82c924a9acf28d623fc6ca727      cat-
me.txt
100644 blob bb2fe940924c65b4a1cefcbdbe88c74d39eb23cd
hello_world.c

```

We can also specify that we want the tree object from the commit pointed to by `HEAD`, by specifying `git cat-file -p HEAD^{tree}`, which would give the same results as the previous one. The special notation `HEAD^{tree}` means that from the reference given, (`HEAD`) recursively dereferences the object at the reference until a tree object is found. The first tree object is the root tree object found from the commit pointed to by the `master` branch, which is pointed to by `HEAD`. A generic form of the notation is `<rev>^<type>` and will return the first object of `<type>` searching recursively from `<rev>`.

From the tree object, we can see what it contains: file typePermissions, type (tree/blob), ID, and pathname:

Type/ Permissions	Type	ID/SHA-1	Pathname
100644	blob	f21dc2804e888fee6014d7e5b1ceee533b222c15	README.md
040000	tree	abc267d04fb803760b75be7e665d3d69eed32f8	a_sub_directory
100644	blob	b50f80ac4d0a36780f9c0636f43472962154a11a	another-file.txt
100644	blob	92f046f17079aa82c924a9acf28d623fc6ca727	cat-me.txt
100644	blob	bb2fe940924c65b4a1cefcbdbe88c74d39eb23cd	hello-world.c

The blob object

Now, we can investigate the blob (file) object. We can do it using the same command, giving the blob ID as target for the `cat-me.txt` file:

```
$ git cat-file -p 92f046f17079aa82c924a9acf28d623fc6ca727
```

This is the content of the file: "cat-me.txt."

```
Not really that exciting, huh?
```

This is simply the content of the file, which we will also get by running a normal `cat cat-me.txt` command. So, the objects are tied together, blobs to trees, trees to other trees, and the root tree to the commit object, all by the SHA-1 identifier of the object.

The branch

The branch object is not really like any other Git objects; you can't print it using the `cat-file` command as we can with the others (if you specify the `-p` pretty print, you'll just get the commit object it points to):

```
$ git cat-file master
usage: git cat-file (-t|-s|-e|-p|<type>|--textconv) <object>
      or: git cat-file (--batch|--batch-check) <<list_of_objects>

<type> can be one of: blob, tree, commit, tag.
...
$ git cat-file -p master
tree 34fa038544bcd9aed660c08320214bafff94150b
parent a90d1906337a6d75f1dc32da647931f932500d83
...
```

Instead, we can take a look at the branch inside the `.git` folder where the whole Git repository is stored. If we open the text file `.git/refs/heads/master`, we can actually see the commit ID the `master` branch points to. We can do this using `cat` as follows:

```
$ cat .git/refs/heads/master
34acc370b4d6ae53f051255680feaefaf7f7850d
```

We can verify that this is the latest commit by running `git log -1`:

```
$ git log -1
commit 34acc370b4d6ae53f051255680feaefaf7f7850d
Author: Aske Olsson <aske.olsson@switch-gears.dk>
Date:   Fri Dec 13 12:26:00 2013 +0100
```

This is the subject line of the commit message

```
...
```

We can also see that `HEAD` is pointing to the active branch by using `cat` with the `.git/HEAD` file:

```
$ cat .git/HEAD  
ref: refs/heads/master
```

The branch object is simply a pointer to a commit, identified by its SHA-1 hash.

The tag object

The last object to be analyzed is the `tag` object. There are three different kinds of tags: a lightweight (just a label) tag, an annotated tag, and a signed tag. In the example repository, there are two annotated tags:

```
$ git tag  
v0.1  
v1.0
```

Let's take a closer look at the `v1.0` tag:

```
$ git cat-file -p v1.0  
object 34acc370b4d6ae53f051255680feaefaf7f7850d  
type commit  
tag v1.0  
tagger Aske Olsson <aske.olsson@switch-gears.dk> 1386941492  
+0100
```

We got the hello world C program merged, let's call that a release 1.0

As you can see, the tag consists of an object, which in this case is the latest commit on the master branch, the object's type (both, commits, and blobs and trees can be tagged), the tag name, the tagger and timestamp, and finally a tag message.

How it works...

The Git command `git cat-file -p` will pretty print the object given as an input. Normally, it is not used in everyday Git commands, but it is quite useful to investigate how it ties together the objects. We can also

verify the output of `git cat-file`, by rehashing it with the Git command `git hash-object`; for example, if we want to verify the commit object at `HEAD` (`34acc370b4d6ae53f051255680feaefaf7f7850d`), we can run the following command:

```
$ git cat-file -p HEAD | git hash-object -t commit --stdin  
34acc370b4d6ae53f051255680feaefaf7f7850d
```

If you see the same commit hash as `HEAD` pointing towards you, you can verify whether it is correct with `git log -1`.

There's more...

There are many ways to see the objects in the Git database. The `git ls-tree` command can easily show the contents of trees and subtrees and `git show` can show the Git objects, but in a different way.

See also

- For further information about Git plumbing, see [Chapter 11, *Git Plumbing and Attributes*](#), almost at the end of this book.

The three stages

We have seen the different objects in Git but how do we create them? In this example, we'll see how to create a blob, tree, and commit object in the repository. We'll learn about the three stages of creating a commit.

Getting ready

We'll use the same `data-model` repository as seen in the last recipe:

```
$ git clone https://github.com/dvaske/data-model.git  
$ cd data-model
```

How to do it...

First, we'll make a small change to the file and check `git status`:

```
$ echo "Another line" >> another-file.txt  
$ 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:   another-file.txt  
  
no changes added to commit (use "git add" and/or "git commit -a")
```

This, of course, just tells us that we have modified `another-file.txt` and we need to use `git add` to stage it. Let's add the `another-file.txt` file and run `git status` again:

```
$ git add another-file.txt  
$ 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: another-file.txt
```

The file is now ready to be committed, just as you have probably seen before. But what happened during the `add` command? The `add` command, generally speaking, moves files from the working directory to the staging area, but more than this actually happens, though you don't see it. When a file is moved to the staging area, the SHA-1 hash of the file is created and the blob object is written to Git's database. This happens for all the files added and every time a file is added, but if nothing changes for a file, this means it is already stored in the database. At first, this might seem that the database is growing quickly, but this is not the case. Garbage collection kicks in at times, compressing and cleaning up the database and keeping only the objects that are required.

We can edit the file again and run `git status`:

```
$ echo 'Whoops almost forgot this' >> another-file.txt
$ 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: another-file.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: another-file.txt
```

Now, the file shows up both in the **Changes to be committed** and **Changes not staged for commit** sections. This looks a bit weird at first, but there is of course an explanation. When we added the file the first time, the content of it was hashed and stored in Git's database. The changes from the second change of the file have not yet been hashed and written to the database; it only exists in the working directory.

Therefore, the file shows up in both the **Changes to be committed** and **Changes not staged for commit** sections; the first change is ready to be committed, the second is not. Let's also add the second change:

```
$ git add another-file.txt  
$ 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:   another-file.txt
```

Now, all the changes we have made to the file are ready to be committed and we can record a commit:

```
$ git commit -m 'Another change to another file'  
[master 55e29e4] Another change to another file  
 1 file changed, 2 insertions(+)
```

How it works...

As we learned previously, the `add` command creates the blob object, the tree, and commit objects; however, they are created when we run the `commit` command. We can view these objects with the `cat-file` command, as we saw in the previous recipe:

```
$ git cat-file -p HEAD  
tree 162201200b5223d48ea8267940c8090b23cbfb60  
parent 34acc370b4d6ae53f051255680feaefaf7f7850d  
author Aske Olsson <aske@schantz.com> 1401744547 +0200  
committer Aske Olsson <aske@schantz.com> 1401744547 +0200
```

```
Another change to another file
```

The root-tree object from the commit is:

```
$ git cat-file -p HEAD^{tree}  
100644 blob f21dc2804e888fee6014d7e5b1ceee533b222c15  README.md  
040000 tree abc267d04fb803760b75be7e665d3d69eed32f8  
a_sub_directory  
100644 blob 35d31106c5d6fdb38c6b1a6fb43a90b183011a4b  another-
```

```
file.txt
100644 blob 92f046f17079aa82c924a9acf28d623fc6ca727 cat-
me.txt
100644 blob bb2fe940924c65b4a1cefcbdbe88c74d39eb23cd
hello_world.c
```

From the previous recipe, we know the SHA-1 of the root tree was 34fa038544bcd9aed660c08320214bafff94150b and of the another-file.txt file was b50f80ac4d0a36780f9c0636f43472962154a11a, and as expected, they changed in our latest commit when we updated the another-file.txt file. We added the same file, another-file.txt, twice before we created the commit, recording the changes to the history of the repository. We also learned that the `add` command creates a blob object when called. So in the Git database, there must be an object similar to the content of `another-file.txt` the first time we added the file to the staging area. We can use the `git fsck` command to check for dangling objects, that is, objects that are not referred by other objects or references:

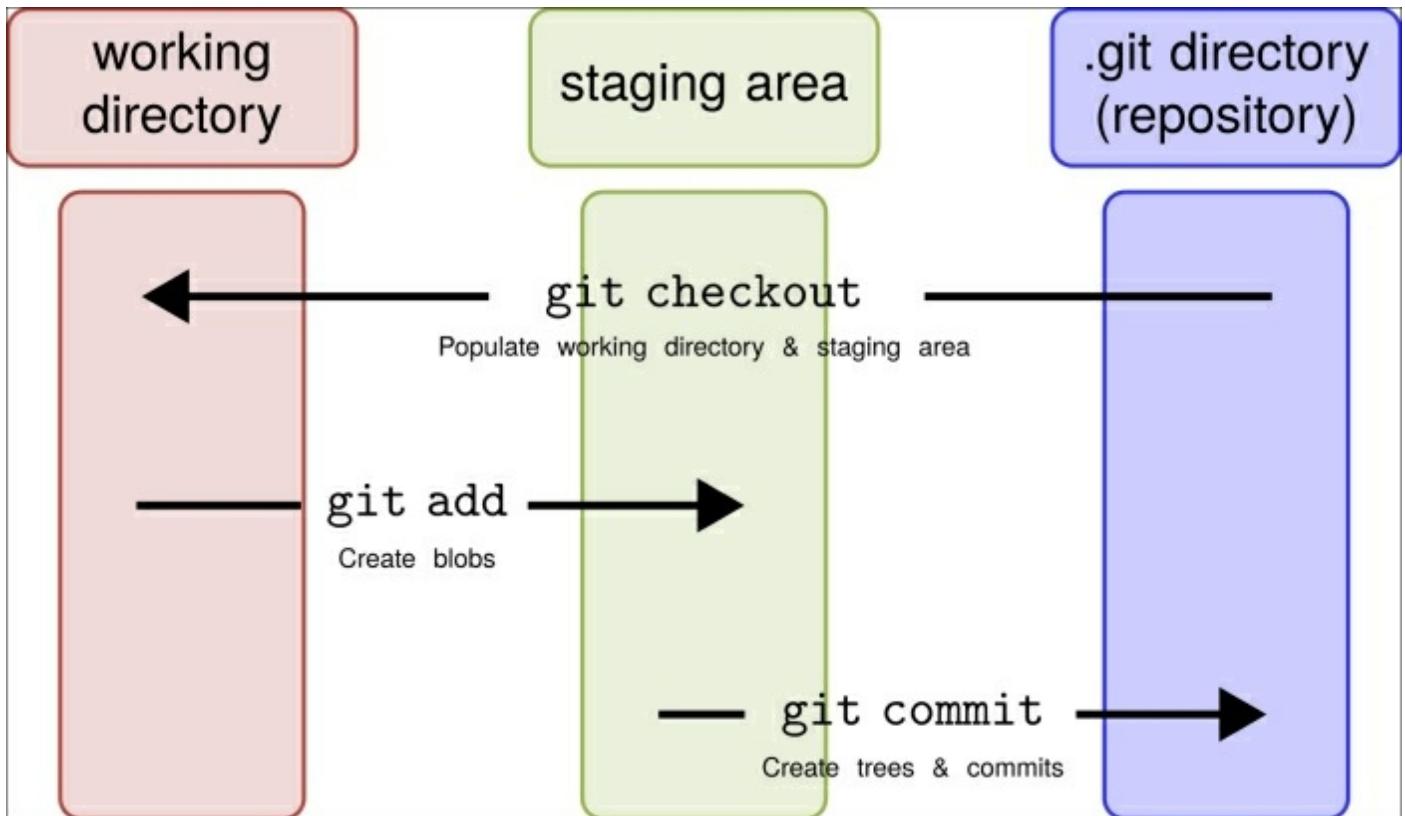
```
$ git fsck --dangling
Checking object directories: 100% (256/256), done.
dangling blob ad46f2da274ed6c79a16577571a604d3281cd6d9
```

Let's check the contents of the blob using the following command:

```
$ git cat-file -p ad46f2da274ed6c79a16577571a604d3281cd6d9
This is just another file
Another line
```

The blob is, as expected, similar to the content of `another-file.txt` when we added it to the staging area the first time.

The following diagram describes the tree stages and the commands used to move between the stages:

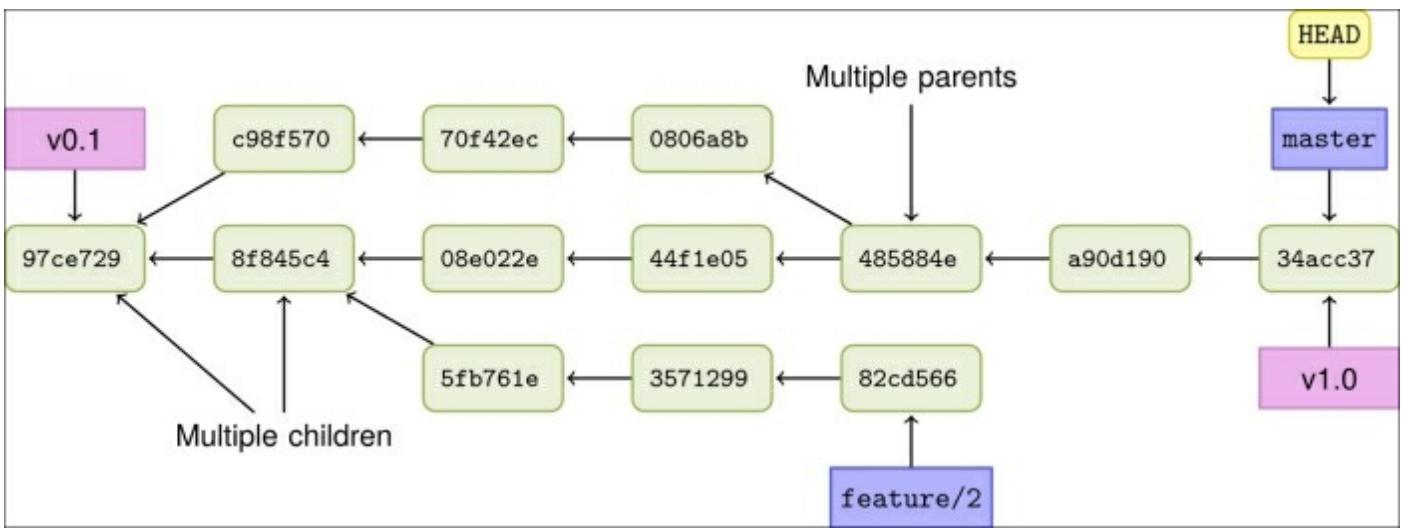


See also

- For more examples and information on the `cat-file`, `fsck`, and other plumbing commands, see [Chapter 11, Git Plumbing and Attributes](#).

Viewing the DAG

The history in Git is formed from the commit objects; as development advances, branches are created and merged, and the history will create a directed acyclic graph, the DAG, due to the way Git ties a commit to its parent commit. The DAG makes it easy to see the development of a project based on the commits. Please note that the arrows in the following diagram are dependency arrows, meaning that each commit points to its parent commit(s), hence the arrows point in the opposite direction of time:



A graph of the example repository with abbreviated commit IDs

Viewing the history (the DAG) is built into Git by its `git log` command. There are also a number of visual Git tools that can graphically display the history. This section will show some features of `git log`.

Getting ready

We will use the example repository from the last section and ensure that the master branch is pointing to `34acc37`:

```
$ git checkout master && git reset --hard 34acc37
```

In the previous command, we only use the first seven characters (34acc37) of the commit ID; this is fine as long as the abbreviated ID used is unique in the repository.

How to do it...

The simplest way to see the history is to use the `git log` command; this will display the history in reverse chronological order. The output is paged through `less` and can be further limited, for example, by providing only the number of commits in history to be displayed:

```
$ git log -3
```

This will display the following result:

```
commit 34acc370b4d6ae53f051255680feaefaf7f7850d
Author: Aske Olsson <aske.olsson@switch-gears.dk>
Date:   Fri Dec 13 12:26:00 2013 +0100
```

This is the subject line of the commit message.

It should be followed by a blank line then the body, which is this text. Here

you can have multiple paragraphs etc. and explain your commit. It's like an

email with subject and body, so get people's attention in the subject

```
commit a90d1906337a6d75f1dc32da647931f932500d83
Author: Aske Olsson <aske.olsson@switch-gears.dk>
Date:   Fri Dec 13 12:17:42 2013 +0100
```

Instructions for compiling `hello_world.c`

```
commit 485884efd6ac68cc7b58c643036acd3cd208d5c8
Merge: 44f1e05 0806a8b
Author: Aske Olsson <aske.olsson@switch-gears.dk>
Date:   Fri Dec 13 12:14:49 2013 +0100
```

Merge branch 'feature/1'

Adds a hello world C program.

Tip

Turn on colors in the Git output by running `git config --global color.ui auto`.

By default, `git log` prints the commit, author's name and e-mail ID, timestamp, and the commit message. However, the information isn't very graphical, especially if you want to see branches and merges.

To display this information and limit some of the other data, you can use the following options with `git log`:

```
$ git log --decorate --graph --oneline --all
```

The previous command will show one commit per line (`--oneline`) identified by its abbreviated commit ID and the commit message subject. A graph will be drawn between the commits depicting their dependency (`--graph`). The `--decorate` option shows the branch names after the abbreviated commit ID, and the `--all` option shows all the branches, instead of just the current one(s).

```
$ git log --decorate --graph --oneline --all
* 34acc37 (HEAD, tag: v1.0, origin/master, origin/HEAD, master)
This is the sub...
* a90d190 Instructions for compiling hello_world.c
* 485884e Merge branch 'feature/1'
...
```

This output, however, gives neither the timestamp nor author information, due to the way the `--oneline` option formats the output.

Fortunately, the `log` command gives us the possibility to create our own output format. So, we can make a history view similar to the previous. The colors are made with the `%C<color-name>text-be-colored%Creset` syntax: including the author and timestamp information, and some colors to display it nicely:

```
$ git log --all --graph --pretty=format:\
'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%ci) %C(bold blue)<%an>%Creset'
```

```

dwaske@w510: ~/Documents/packt/repos/data-model (master)$ git log --all --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%ci) %C(bold blue)<%an>%Creset'
* 34acc37 - (HEAD, tag: v1.0, origin/master, master) This is the subject line of the commit message (2013-12-13 14:30:55 +0100) <Aske Olsson>
* a90d190 - Instructions for compiling hello_world.c (2013-12-13 12:17:42 +0100) <Aske Olsson>
* 485884e - Merge branch 'feature/1' (2013-12-13 12:14:49 +0100) <Aske Olsson>
| \
| * 0806a8b - Fixes warnings with -Wall flag (2013-12-13 12:14:35 +0100) <Aske Olsson>
| * 70f42ec - Remove warnings when compiling (2013-12-13 12:13:07 +0100) <Aske Olsson>
| * c98f570 - Adds a hello world C program (2013-12-13 12:12:37 +0100) <Aske Olsson>
| * 44f1e05 - Add another file to the repository (2013-12-13 12:07:41 +0100) <Aske Olsson>
| * 08e022e - More master info in README (2013-12-13 11:42:25 +0100) <Aske Olsson>
| * 82cd566 - (origin/feature/2, feature/2) Makes hello_world python & perl executable (2013-12-13 12:37:40 +0100) <Aske Olsson>
| * 3571299 - Adds python and perl hello worlds (2013-12-13 12:35:02 +0100) <Aske Olsson>
| * 5fb761e - Adds Java hello world (2013-12-13 12:30:36 +0100) <Aske Olsson>
|
|/
* 8f845c4 - Update README with master branch info (2013-12-13 11:42:01 +0100) <Aske Olsson>
|
* 97ce729 - (tag: v0.1) Initial commit for data-model repository (2013-12-13 11:41:03 +0100) <Aske Olsson>
dwaske@w510: ~/Documents/packt/repos/data-model (master)$

```

This is a bit cumbersome to write, but luckily it can be made as an alias so you only have to write it once:

```
git config ----global alias.graph "log --all --graph --
pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s
%Cgreen(%ci) %C(bold blue)<%an>%Creset'"
```

Tip

Now, all you need to do is call `git graph` to show the history as you saw previously.

How it works...

Git traverses the DAG by following the parent IDs (hashes) from the given commit(s). The options passed to `git log` can format the output in different ways; this can serve several purposes, for example, to give a nice graphical view of the history, branches, and tags, as seen previously, or to extract specific information from the history of a repository to use, for example, in a script.

See also

- For more information about configuration and aliases, see [Chapter 2, Configuration](#).

Extracting fixed issues

A common use case when creating a release is to create a release note, containing among other things, the bugs fixed in the release. A good practice is to write in the commit message if a bug is fixed by the commit. A better practice is to have a standard way of doing it, for example, a line with the string "Fixes-bug: " followed by the bug identifier in the last part of the commit message. This makes it easy to compile a list of bugs fixed for a release note. The JGit project is a good example of this; their bug identifier in the commit messages is a simple "Bug: " string followed by the bug ID.

This recipe will show you how to limit the output of `git log` to list just the commits since the last release (tag), which contains a bug fix.

Getting ready

Clone the JGit repository using the following command lines:

```
$ git clone https://git.eclipse.org/r/jgit/jgit  
$ cd jgit
```

If you want the exact same output as in this example, reset your `master` branch to the following commit,

`b14a93971837610156e815ae2eee3baaa5b7a44b`:

```
$ git checkout master && git reset --hard b14a939
```

How to do it...

You are now ready to look through the commit log for commit messages that describe the bugs fixed. First, let's limit the log to only look through the history since the last tag (release). To find the last tag, we can use `git describe`:

```
$ git describe  
v3.1.0.201310021548-r-96-gb14a939
```

The preceding output tells us three things:

- The last tag was v3.1.0.201310021548-r
- The number of commits since the tag were 96
- The current commit in abbreviated form is b14a939

Now, the log can be parsed from HEAD to v3.1.0.201310021548-r. But just running `git log 3.1.0.201310021548-r..HEAD` will give us all the 96 commits, and we just want the commits with commit messages that contain "Bug: xxxxxxx" for our release note. The xxxxxxx is an identifier for the bug, for example, a number. We can use the --grep option with `git log` for this purpose: `git log --grep "Bug: "`. This will give us all the commits with "Bug: " in the commit message; all we need now is just to format it to something we can use for our release note.

Let's say we want the release note format to look like the following template:

```
Commit-id: Commit subject  
Fixes-bug: xxx
```

Our command line so far is as follows:

```
$ git log --grep "Bug: " v3.1.0.201310021548-r..HEAD
```

This gives us all the bug fix commits, but we can format this to a format that is easily parsed with the --pretty option. First, we will print the abbreviated commit ID %h, followed by a separator of our choice |, then the commit subject %s, (first line of the commit message), followed by a new line %n, and the body, %b:

```
--pretty="%h|%s%n%b"
```

The output of course needs to be parsed, but that's easy with regular Linux tools such as grep and sed:

First, we just want the lines that contain "|" or "Bug: ":

```
grep -E "\||Bug: "
```

Then, we replace these with `sed`:

```
sed -e 's/|/: /' -e 's/Bug:/Fixes-bug:/'
```

The entire command put together gives:

```
\$ git log --grep "Bug: " v3.1.0.201310021548-r..HEAD --
pretty="%h|%s%n%b"
| grep -E "\||Bug: " | sed -e 's/|/: /' -e 's/Bug:/Fixes-bug:/'
```

The previous set of commands gives the following output:

```
f86a488: Implement rebase.autostash
Fixes-bug: 422951
7026658: CLI status should support --porcelain
Fixes-bug: 419968
e0502eb: More helpful InvalidPathException messages (include
reason)
Fixes-bug: 413915
f4dae20: Fix IgnoreRule#isMatch returning wrong result due to
missing reset
Fixes-bug: 423039
7dc8a4f: Fix exception on conflicts with recursive merge
Fixes-bug: 419641
99608f0: Fix broken symbolic links on Cygwin.
Fixes-bug: 419494
...
```

Now, we can extract the bug information from the bug tracker and put the preceding code in the release note as well, if necessary.

How it works...

First, we limit the `git log` command to only show the range of commits we are interested in, then we further limit the output by filtering the "Bug: " string in the commit message. We pretty print the string so we can easily format it to a style we need for the release note and finally find and replace with `grep` and `sed` to completely match the style of the release note.

There's more...

If we just wanted to extract the bug IDs from the commit messages and didn't care about the commit IDs, we could have just used `grep` after the `git log` command, still limiting the log to the last tag:

```
$ git log v3.1.0.201310021548-r..HEAD | grep "Bug: "
```

If we just want the commit IDs and their subjects but not the actual bug IDs, we can use the `--oneline` feature of `git log` combined with the `--grep` option:

```
$ git log --grep "Bug: " --oneline v3.1.0.201310021548-r..HEAD
```

Getting a list of the changed files

As seen in the previous recipe where a list of fixed issues was extracted from the history, a list of all the files that have been changed since the last release can also easily be extracted. The files can be further filtered to find those that have been added, deleted, modified, and so on.

Getting ready

The same repository and `HEAD` position (`HEAD` pointing to `b14a939`) as seen in the previous recipe will be used. The release is also the same, which is `v3.1.0.201310021548-r`.

How to do it...

The following command lists all the files changed since the last release (`v3.1.0.201310021548-r`):

```
$ git diff --name-only v3.1.0.201310021548-r..HEAD
org.eclipse.jgit.packaging/org.eclipse.jgit.target/jgit-
4.3.target
org.eclipse.jgit.packaging/org.eclipse.jgit.target/jgit-
4.4.target
org.eclipse.jgit.pgm.test/tst/org/eclipse/jgit/pgm/DescribeTest
.java
org.eclipse.jgit.pgm.test/tst/org/eclipse/jgit/pgm/FetchTest.ja
va
org.eclipse.jgit.pgm/src/org/eclipse/jgit/pgm/Describe.java
...
```

How it works...

The `git diff` command operates on the same revision range as `git log` did in the previous recipe. By specifying `--name-only`, Git will only give the paths of the files as output changed by the commits in the range specified.

There's more...

The output of the command can be further filtered; if we only want to show which files have been deleted in the repository since the last commit, we can use the `--diff-filter=D` switch with `git diff`:

```
$ git diff --name-only --diff-filter=D v3.1.0.201310021548-r..HEAD  
org.eclipse.jgit.junit/src/org/eclipse/jgit/junit/SampleDataRep  
ositoryTestCase.java  
org.eclipse.jgit.packaging/org.eclipse.jgit.target/org.eclipse.  
jgit.target.target  
org.eclipse.jgit.test/tst/org/eclipse/jgit/internal/storage/fil  
e/GCTest.java
```

There are also switches for the files that have been added (`A`), copied (`C`), deleted (`D`), modified (`M`), renamed (`R`), and so on.

See also

- For more information, visit the help page by running `git help diff`

Viewing history with Gitk

We saw earlier how we can view the history (the DAG) and visualize it with the use of `git log`. However, as the history grows, the terminal representation of the history can be a bit cumbersome to navigate. Fortunately, there are a lot of graphical tools around Git, one of them being Gitk, which works on multiple platforms (Linux, Mac, and Windows).

This recipe will show you how to get started with Gitk.

Getting ready

Make sure you have Gitk installed:

```
$ which gitk  
/usr/local/bin/gitk
```

If nothing shows up, Gitk is not installed on your system, or at least is not available on your `$PATH`.

Change the directory to the `data-model` repository from the objects and DAG examples. Make sure the master branch is checked out and pointing to `34acc37`:

```
$ git checkout master && git reset --hard 34acc37
```

How to do it...

In the repository, run `gitk --all &` to bring up the Gitk interface. You can also specify the commit range or branches you want similar to `git log` or provide `--all` to see everything:

```
$ gitk --all &
```

The screenshot shows the Gitk graphical interface for Git. The top half displays a commit history graph with nodes for branches like 'master' and 'feature/1'. A specific commit node for 'v0.1' is highlighted. The bottom half provides a detailed view of this commit, including its author ('Aske Olsson'), committer ('Aske Olsson'), date ('2013-12-13 11:42:01'), subject ('Initial commit for data-model-repository'), and body. It also shows the commit message and the patch applied to a file named 'cat-me.txt'.

```

graph TD
    master[v1.0] --- feature1[feature/1]
    master --- v0_1[v0.1]
    feature1 --- v0_1
    v0_1 --- commit1[Instructions for compiling hello_world.c]
    v0_1 --- commit2[Merge branch 'feature/1']
    v0_1 --- commit3[Fixes warnings with -Wall flag]
    v0_1 --- commit4[Remove warnings when compiling]
    v0_1 --- commit5[Adds a hello world C program]
    v0_1 --- commit6[Add another file to the repository]
    v0_1 --- commit7[More master info in README]
    v0_1 --- commit8[feature/2]
    v0_1 --- commit9[remotes/origin/feature/2]
    v0_1 --- commit10[ Makes hello_wor]
    v0_1 --- commit11[ Adds python and perl hello worlds]
    v0_1 --- commit12[ Adds Java hello world]
    v0_1 --- commit13[Update README with master branch info]
    v0_1 --- commit14[Initial commit for data-model-repository]

Commit v0.1 details:
Author: Aske Olsson <aske.olsson@switch-gears.dk> 2013-12-13 11:42:01 +0000
Committer: Aske Olsson <aske.olsson@switch-gears.dk> 2013-12-13 11:42:01 +0000
Tags: v1.0
Parent: a90d1906337a6d75f1dc32da647931f932500d83 (Instructions for compiling hello_world.c)
Branches: master, remotes/origin/master
Follows: v0.1
Precedes:

This is the subject line of the commit message

It should be followed by a blank line then the body,
you can have multiple paragraphs etc. and explain your
email with subject and body, so get people's attention

----- a_sub_directory/readme -----
new file mode 100644
index 000000..6dc3bfb
@@ -0,0 +1 @@
+A file in a sub directory

```

How it works...

Gitk parses the information for every commit and the objects attached to it to provide an easy graphical information screen that shows a graph of the history, author, and timestamp for each commit. In the bottom half, the commit message and the patches for each file changed and the list of files changed by the selected commit are displayed.

Though very lightweight and fast, Gitk is a very powerful tool. There are many different context menus regarding clicking on a commit, a branch, or a tag in the history view. You can create and delete branches, revert and cherry-pick commits, diff selected commits, and much more.

There's more...

From the interface, you can perform a find and search. Find looks through the history and search looks through the information displayed in the lower half of Gitk for the currently highlighted commit.

Finding commits in history

You already saw in the previous recipe how we can filter the output of `git log` to only list commits with the string "Bug: " in the commit message. In this example, we will use the same technique to find specific commits in the entire history.

Getting ready

Again, we will use the JGit repository, trying to find commits related to the keyword "Performance". In this recipe, we will look through the entire history, so we don't need the master branch to point to a specific commit.

How to do it...

As we tried earlier, we can use the `--grep` option to find specific strings in commit messages. In this recipe, we look at the entire history and search every commit that has "Performance" in its commit message:

```
$ git log --grep "Performance" --oneline --all
9613b04 Merge "Performance fixes in DateRevQueue"
84afea9 Performance fixes in DateRevQueue
7cad0ad DHT: Remove per-process ChunkCache
d9b224a Delete DiffPerformanceTest
e7a3e59 Reuse DiffPerformanceTest support code to validate
algorithms
fb1c7b1 Wait for JIT optimization before measuring diff
performance
```

How it works...

In this example, we specifically ask Git to consider all of the commits in the history, by supplying the `--all` switch. Git runs through the DAG and checks whether the "Performance" string is included in the commit message. For an easy overview of the results, the `--oneline` switch is also used to limit the output to just the subject of the commit message. Hopefully then the commit(s) we needed to find can be identified from

this much shorter list of commits.

Note that the search is case sensitive; had we searched for "performance" (all in lower case), the list of commits would have been very different:

```
$ git log --grep "performance" --oneline --all
5ef6d69 Use the new FS.exists method in commonly occurring
places
2be6927 Always allocate the PackOutputStream copyBuffer
437be8d Simplify UploadPack by parsing wants separately from
haves
e6883df Enable writing bitmaps during GC by default.
374406a Merge "Fix RefUpdate performance for existing Refs"
f1dea3e Fix RefUpdate performance for existing Refs
84afea9 Performance fixes in DateRevQueue
8a9074f Implement core.checkstat = minimal
130ad4e Delete storage.dht package
d4fed9c Refactored method to find branches from which a commit
is reachable
...
```

There's more...

We also could have used the find feature in Githk to find the same commits. Open Githk with the `--all` switch, type `Performance` in the **Find** field and hit *Enter*. This will highlight the commits in the history view and you can navigate to the previous/next result by pressing *Shift + up arrow*, *Shift + down arrow*, or the buttons next to the **Find** field. You will still, however, be able to see the entire history in the view with the matching commits highlighted:

File Edit View Help

Include supported extensions in PackFile constructor.
 Fix while boundaries in DateRevQueue.add()
Merge "Performance fixes in DateRevQueue"
Performance fixes in DateRevQueue
 Update last release version to 2.3.1.201302201838-r
 Deploy Maven artifacts to Eclipse Nexus repository
 Implement recursive merge strategy
 Fix off by one error in PackReverseIndex.
 Merge branch 'stable-2.3'
remotes/origin/stable-2.3 Prepare 2.3.2-SNAPSHOT bu
v2.3.1.201302201838-r JGit v2.3.1.201302201838-r
 Merge "Accept Change-Id even if footer contains not well-
 Accept Change-Id even if footer contains not well-formed
 Fix false positives in hashing used by PathFilterGroup

SHA1 ID: 84afea9179932995d1e59f8fda4e6b11217382ad ← → Row: 451 / 2897

Find next prev commit containing: Performance

Search Lines of context: 3 Ignore space change Line diff

Author: Gustaf Lundh <gustaf.lundh@sonymobile.com> 2013-02-25 18:24:16
 Committer: Gustaf Lundh <gustaf.lundh@sonymobile.com> 2013-02-25 18:24:16
 Parent: [51d0e1f26e23d04ae73054958546159e01196a4d](#) (Fix Con
 Child: [9613b04d8143c74e729acda414e6392078297d33](#) (Merge "
 Branches: [master](#), [remotes/origin/master](#),
[remotes/origin/stable-3.0](#), [remotes/origin/stable-3.1](#),
[remotes/origin/stable-3.2](#)
 Follows: [v2.2.0.201212191850-r](#)
 Precedes: [v3.0.0.201305080800-m7](#)

Performance fixes in DateRevQueue

When a lot of commits are added to DateRevQueue, the

Comments
 org.eclipse.jgit/src/org/eclipse/jgit/revwalk/DateRevQueue.java

Patch Tree

Searching through history code

Sometimes it is not enough; by just looking through the commit messages in the history, you may want to know which commits touched a specific method or variable. This is also possible using `git log`. You can perform a search for a string, for example, a variable or method, and `git log` will give you the commits, adding or deleting the string from the history. In this way, you can easily get the full commit context for the piece of code.

Getting ready

Again, we will use the JGit repository with the master branch pointing to b14a939:

```
$ git checkout master && git reset --hard b14a939
```

How to do it...

We would like to find all the commits that have changes made to lines that contain the method "isOutdated". Again, we will just display the commits on one line each then we can check them individually later:

```
$ git log -G"isOutdated" --oneline
f32b861 JGit 3.0: move internal classes into an internal
subpackage
c9e4a78 Add isOutdated method to DirCache
797ebba Add support for getting the system wide configuration
ad5238d Move FileRepository to storage.file.FileRepository
4c14b76 Make lib.Repository abstract and lib.FileRepository its
implementation
c9c57d3 Rename Repository 'config' as 'repoConfig'
5c780b3 Fix unit tests using MockSystemReader with user
configuration
cc905e7 Make Repository.getConfig aware of changed config
```

Eight commits have patches that involve the string "isOutdated".

How it works...

Git traverses the history, the DAG, looking at each commit for the string "isOutdated" in the patch between the parent commit and the current commit. This method is quite convenient to find out when a given string was introduced or deleted and to get the full context and commit at that point in time.

There's more...

The `-G` option used with `git log` will look for differences in the patches that contain added or deleted lines that match the given string. However, these lines could also have been added or removed due to some other refactoring/renaming of a variable or method. There is another option that can be used with `git log`, `-S`, which will look through the difference in the patch text similar to the `-G` option, but only match commits where there is a change in the number of occurrences of the specified string, that is, a line added or removed, but not added and removed.

Let's see the output of the `-S` option:

```
$ git log -S"isOutdated" --oneline
f32b861 JGit 3.0: move internal classes into an internal
subpackage
c9e4a78 Add isOutdated method to DirCache
797ebba Add support for getting the system wide configuration
ad5238d Move FileRepository to storage.file.FileRepository
4c14b76 Make lib.Repository abstract and lib.FileRepository its
implementation
5c780b3 Fix unit tests using MockSystemReader with user
configuration
cc905e7 Make Repository.getConfig aware of changed config
```

The search matches seven commits, whereas the search with the `-G` option matches eight commits. The difference is the commit with the ID `c9c57d3` is only found with the `-G` option in the first list. A closer look at this commit shows that the `isOutdated` string is only touched due to renaming of another object, and this is why it is filtered away from the list of matching commits in the last list when using the `-S` option. We can see the content of the commit with the `git show` command, and use `grep`

`-C4` to limit the output to just the four lines before and after the search string:

```
$ git show c9c57d3 | grep -C4 "isOutdated"
@@ -417,14 +417,14 @@ public FileBasedConfig getConfig() {
        throw new RuntimeException(e);
    }
}
-
- if (config.isOutdated()) {
+ if (repoConfig.isOutdated()) {
    try {
        loadConfig();
        loadRepoConfig();
    } catch (IOException e) {
```

Chapter 2. Configuration

In this chapter, we will cover the following topics:

- Configuration targets
- Querying the existing configuration
- Templates
- A .git directory template
- A few configuration examples
- Git aliases
- The refspec exemplified

Configuration targets

In this section, we will look at the different layers that can be configured. The layers are:

- SYSTEM: This layer is system-wide and found in `/etc/gitconfig`
- GLOBAL: This layer is global for the user and found in `~/.gitconfig`
- LOCAL: This layer is local to the current repository and found in `.git/config`

Getting ready

We will use the `jgit` repository for this example; clone it or use the clone you already have from [Chapter 1](#), *Navigating Git*, as shown in the following command:

```
$ git clone https://git.eclipse.org/r/jgit/jgit
$ cd jgit
```

How to do it...

In the previous example, we saw how we could use the command `git config --list` to list configuration entries. This list is actually made from three different levels of configuration that Git offers: system-wide configuration, SYSTEM; global configuration for the user, GLOBAL; and local repository configuration, LOCAL.

For each of these configuration layers, we can query the existing configuration. On a Windows box with a default installation of the Git extensions, the different configuration layers will look approximately like the following:

```
$ git config --list --system
core.symlinks=false
core.autocrlf=true
color.diff=auto
color.status=auto
color.branch=auto
color.interactive=true
pack.pack sizelimit=2g
help.format=html
http.sslcainfo=/bin/curl-ca-bundle.crt
sendemail.smtpserver=/bin/mssmtp.exe
diff.astextplain.textconv=astextplain
rebase.autosquash=true

$ git config --list --global
merge.tool=kdiff3
mergetool.kdiff3.path=C:/Program Files (x86)/KDiff3/kdiff3.exe
diff.guitool=kdiff3
difftool.kdiff3.path=C:/Program Files (x86)/KDiff3/kdiff3.exe
core.editor="C:/Program Files (x86)/GitExtensions/GitExtensions.exe" fileeditor
core.autocrlf=true
credential.helper=!\"C:/Program Files (x86)/GitExtensions/GitCredentialWinStore/git-credential-winstore.exe\"
user.name=Aske Olsson
user.email=aske.olsson@switch-gears.dk
$ git config --list --local
core.repositoryformatversion=0
core.filemode=false
core.bare=false
core.logallrefupdates=true
core.symlinks=false
core.ignorecase=true
core.hidedotfiles=dotGitOnly
remote.origin.url=https://git.eclipse.org/r/jgit/jgit
remote.origin.fetch=+refs/heads/*:refs/remotes/origin/*
branch.master.remote=origin
branch.master.merge=refs/heads/master
```

We can also query a single key and limit the scope to one of the three layers, by using the following command:

```
$ git config --global user.email  
aske.olsson@switch-gears.dk
```

We can set the e-mail address of the user to a different one for the current repository:

```
$ git config --local user.email aske@switch-gears.dk
```

Now, listing the GLOBAL layer `user.email` will return `aske.olsson@switch-gears.dk`, listing LOCAL gives `aske@switch-gears.dk`, and listing `user.email` without specifying the layer gives the effective value that is used in the operations on this repository, in this case, the LOCAL value `aske@switch-gears.dk`. The effective value is the value, which takes precedence when needed. When two or more values are specified for the same key, but on different layers, the lowest layer takes precedence. When a configuration value is needed, Git will first look in the LOCAL configuration. If not found here, the GLOBAL configuration is queried. If it is not found in the GLOBAL configuration, the SYSTEM configuration is used. If none of this works, the default value in Git is used.

In the previous example, `user.email` is specified in both the GLOBAL and LOCAL layers. Hence, the LOCAL layer will be used.

How it works...

Querying the three layers of configuration simply returns the content of the configuration files: `/etc/gitconfig` for system-wide configuration, `~/.gitconfig` for user-specific configuration, and `.git/config` for repository-specific configuration. When not specifying the configuration layer, the returned value will be the effective value.

There's more...

Instead of setting all the configuration values on the command line by

the key value, it is possible to set them by just editing the configuration file directly. Open the configuration file in your favorite editor and set the configuration you need, or use the built-in `git config -e` repository to edit the configuration directly in the Git-configured editor. You can set the editor to the editor of your choice either by changing the `$EDITOR` environment variable or with the `core.editor` configuration target, for example:

```
$ git config --global core.editor vim
```

Querying the existing configuration

In this example, we will look at how we can query the existing configuration and set the configuration values.

Getting ready

We'll use `jgit` again by using the following command:

```
$ cd jgit
```

How to do it...

To view all the effective configurations for the current Git repository, run the following command:

```
$ git config --list
user.name=Aske Olsson
user.email=askeolsson@switch-gears.dk
core.repositoryformatversion=0
core.filemode=false
core.bare=false
core.logallrefupdates=true
remote.origin.url=https://git.eclipse.org/r/jgit/jgit
remote.origin.fetch=+refs/heads/*:refs/remotes/origin/*
branch.master.remote=origin
branch.master.merge=refs/heads/master
```

The previous output will of course reflect the user running the command. Instead of `Aske Olsson` as the name and the e-mail, the output should reflect your settings.

If we are just interested in a single configuration item, we can just query it by its `section.key` or `section.subsection.key`:

```
$ git config user.name
Aske Olsson
$ git config remote.origin.url
```

<https://git.eclipse.org/r/jgit/jgit>

How it works...

Git's configuration is stored in plaintext files, and works like a key-value storage. You can set/query by key and get the value back. An example of the text-based configuration file is shown as follows (from the `jgit` repository):

```
$ cat .git/config
[core]
    repositoryformatversion = 0
    filemode = false
    bare = false
    logallrefupdates = true
[remote "origin"]
    url = https://git.eclipse.org/r/jgit/jgit
    fetch = +refs/heads/*:refs/remotes/origin/*
[branch "master"]
    remote = origin
    merge = refs/heads/master
```

There's more...

It is also easy to set configuration values. Just use the same syntax as you did when querying the configuration, except you need to add an argument to the value. To set a new e-mail address on the `LOCAL` layer, we can execute the following command line:

```
git config user.email askeolsson@example.com
```

The `LOCAL` layer is the default if nothing else is specified. If you require whitespaces in the value, you can enclose the string in quotation marks, as you would do when configuring your name:

```
git config user.name "Aske Olsson"
```

You can even set your own configuration, which does not have any effect on the core Git, but can be useful for scripting/builds and so on:

```
$ git config my.own.config "Whatever I need"
```

List the value

```
$ git config my.own.config  
Whatever I need
```

It is also very easy to delete/unset configuration entries:

```
$ git config --unset my.own.config
```

List the value

```
$ git config my.own.config
```

Templates

In this example, we will see how to create a template commit message that will be displayed in the editor when creating a commit. The template is only for the local user and not distributed with the repository in general.

Getting ready

In this example, we will use the example repository from [Chapter 1, Navigating Git](#):

```
$ git clone https://github.com/dvaske/data-model.git  
$ cd data-model
```

We'll use the following code as a commit message template for commit messages:

Short description of commit

Longer explanation of the motivation for the change

Fixes-Bug: Enter bug-id or delete line

Implements-Requirement: Enter requirement-id or delete line

Save the commit message template in `$HOME/.gitcommitmsg.txt`. The filename isn't fixed and you can choose a filename of your liking.

How to do it...

To let Git know about our new commit message template, we can set the configuration variable `commit.template` to point at the file we just created with that template; we'll do it globally so it is applicable to all our repositories:

```
$ git config --global commit.template $HOME/.gitcommitmsg.txt
```

Now, we can try to change a file, add it, and create a commit. This will bring up our preferred editor with the commit message template

preloaded:

```
$ git commit
```

Short description of commit

Longer explanation of the motivation for the change

Fixes-Bug: Enter bug-id or delete line

Implements-Requirement: Enter requirement-id or delete line

```
# Please enter the commit message for your changes. Lines
# starting
# with '#' will be ignored, and an empty message aborts the
# commit.
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# modified:   another-file.txt
#
~
~
"\.git/COMMIT_EDITMSG" 13 lines, 396 characters
```

We can now edit the message according to our commit and save to complete the commit.

How it works...

When `commit.template` is set, Git simply uses the content of the template file as a starting point for all commit messages. This is quite convenient if you have a commit-message policy as it greatly increases the chances of the policy being followed. You can even have different templates tied to different repositories, since you can just set the configuration at the local level.

A .git directory template

Sometimes, having a global configuration isn't enough. You will also need to trigger the execution of scripts (also known as Git hooks), exclude files, and so on. It is possible to achieve this with the template option set to `git init`. It can be given as a command-line option to `git clone` and `git init`, or as the `$GIT_TEMPLATE_DIR` environment variable, or as the configuration option `init.templatedir`. It defaults to `/usr/share/git-core/templates`. The template option works by copying files in the template directory to the `.git ($GIT_DIR)` folder after it has been created. The default directory contains sample hooks and some suggested exclude patterns. In the following example, we'll see how we can set up a new template directory, and add a commit message hook and exclude file.

Getting ready

First, we will create the template directory. We can use any name we want, and we'll use `~/.git_template`, as shown in the following command:

```
$ mkdir ~/.git_template
```

Now, we need to populate the directory with some template files. This could be a hook or an exclude file. We will create one hook file and an exclude file. The hook file is located in `.git/hooks/name-of-hook` and the exclude file in `.git/info/exclude`. Create the two directories needed, `hooks` and `info`, as shown in the following command:

```
$ mkdir ~/.git_template/{hooks,info}
```

To keep the sample hooks provided by the default template directory (the Git installation), we copy the files in the default template directory to the new one. When we use our newly created template directory, we'll override the default one. So, copying the default files to our template directory will make sure that except for our specific changes, the template directory is similar to the default one, as shown in the

following command:

```
$ cd ~/.git_template/hooks  
$ cp /usr/share/git-core/templates/hooks/* .
```

We'll use the `commit-msg` hook as the example hook:

```
#!/bin/sh  
MSG_FILE="$1"  
echo "\nHi from the template commit-msg hook" >> $MSG_FILE
```

The hook is very simple and will just add `Hi` from the template `commit-msg` hook to the end of the commit message. Save it as `commit-msg` in the `~/.git_template/hooks` directory and make it executable by using the following command:

```
chmod +x ~/.git_template/hooks/commit-msg
```

Now that the commit message hook is done, let's also add an exclude file to the example. The exclude file works like the `.gitignore` file, but is not tracked in the repository. We'll create an exclude file that excludes all the `*.txt` files, as follows:

```
$ echo *.txt > ~/.git_template/info/exclude
```

Now, our template directory is ready for use.

How to do it...

Our template directory is ready and we can use it, as described earlier, as a command-line option, an environment variable or, as in this example, to be set as a configuration:

```
$ git config --global init.templatedir ~/.git_template
```

Now, all Git repositories we create using `init` or `clone` will have the default files of the template directory. We can test if it works by creating a new repository as follows:

```
$ git init template-example  
$ cd template-example
```

Let's try to create a .txt file and see what `git status` tells us. It should be ignored by the exclude file from the template directory:

```
$ echo "this is the readme file" > README.txt  
$ git status
```

The exclude file worked! You can put in the file endings yourself or just leave it blank and keep to the `.gitignore` files.

To test if the `commit-msg` hook also works, let's try to create a commit. First, we need a file to commit. So, let's create that and commit it as follows:

```
$ echo "something to commit" > somefile  
$ git add somefile  
$ git commit -m "Committed something"
```

We can now check the history with `git log`:

```
$ git log -1  
commit 1f7d63d7e08e96dda3da63eadc17f35132d24064  
Author: Aske Olsson <aske.olsson@switch-gears.dk>  
Date:   Mon Jan 6 20:14:21 2014 +0100  
  
        Committed something  
  
        Hi from the template commit-msg hook
```

How it works...

When Git creates a new repository, either via `init` or `clone`, it will copy the files from the template directory to the new repository when creating the directory structure. The template directory can be defined either by a command-line argument, environment variable, or configuration option. If nothing is specified, the default template directory will be used (distributed with the Git installation). By setting the configuration as a `--global` option, the template directory defined will apply to all of the user's (new) repositories. This is a very nice way to distribute the same hooks across repositories, but it also has some drawbacks. As the files in the template directory are only copied to the Git repositories, updates to the template directory do not affect the existing repositories. This can be

solved by running `git init` in each existing repository to reinitialize the repository, but this can be quite cumbersome. Also, the template directory can enforce hooks on some repositories where you don't want them. This is quite easily solved by simply deleting the hook files in `.git/hooks` of that repository.

See also

- For more information on hooks in Git, please refer to [Chapter 7, *Enhancing Your Daily Work with Git Hooks, Aliases, and Scripts*](#)

A few configuration examples

There are configuration targets in the core Git system. In this section, we'll take a closer look at a few of them that might be useful in your daily work.

We'll look at the following three different configuration areas:

- Rebase and merge setup
- Expiry of objects
- Autocorrect

Getting ready

In this exercise, we'll just set a few configurations. We'll use the data model repository from [Chapter 1, Navigating Git](#):

```
$ cd data-model
```

How to do it...

Let's take a closer look at the previously mentioned configuration areas.

Rebase and merge setup

By default, when performing `git pull`, a merge commit will be created if the history of the local branch has diverged from the remote one. However, to avoid all these merge commits, a repository can be configured so it will default to rebase instead of merging when doing `git pull`. Several configuration targets related to the option exist as follows:

- `pull.rebase`: This configuration, when set to `true`, will pull to rebase the current branch on top of the fetched one when performing a `git pull`. It can also be set to `preserve` so that the local merge commit will not be flattened in the rebase, by passing `--preserve-merges` to `git rebase`. The default value is `false` as the configuration is not set. To set this option in your local repository, run the following command:

```
$ git config pull.rebase true
```

- `branch.autosetuprebase`: When this configuration is set to `always`, any new branch created with `<git branch or git checkout` that tracks another branch will be set up to pull to rebase (instead of merge). The valid options are as follows:
 - `never`: This is set to pull to rebase (default)
 - `local`: This is set to pull to rebase for local tracked branches
 - `remote`: This is set to pull to rebase for remote tracked branches
 - `always`: This is set to pull to rebase for all tracked branches

To set this option for all the new branches regardless of tracking remote or local branches, run the following command:

```
$ git config branch.autosetuprebase always
```

- `branch.<name>.rebase`: This configuration, when set to `true`, applies only to the `<name>` branch and tells Git to pull to rebase when performing `git pull` on the given branch. It can also be set to `preserve` so that the local merge commit will not be flattened when running `git pull`. By default, the configuration is not set for any branch. To set the `feature/2` branch in the repository to default to rebase instead of merge, we can run the following command:

```
$ git config branch.feature/2.rebase true
```

Expiry of objects

By default, Git will perform garbage collection on unreferenced objects and clean `reflog` for entries, both of which are older than 90 days. For an object to be referenced, something must point to it; a tree, a commit, a tag, a branch, or some of the internal Git bookkeeping like `stash` or `reflog`. There are three settings that can be used to change this time as follows:

- `gc.reflogexpire`: This is the general setting to know for how long a branch's history is kept in `reflog`. The default time is 90 days. The setting is a length of time, for example, `10 days`, `6 months` and it can be turned completely off with the value `never`. The setting can

be set to match a `refs` pattern by supplying the pattern in the configuration setting. `gc.<pattern>.reflogexpire`: This pattern can, for example, be `/refs/remotes/*` and the expire setting would then only apply for those refs.

- `gc.reflogexpireunreachable`: This setting controls how long the `reflog` entries that are not a part of the current branch history should be available in the repository. The default value is 30 days, and similar to the previous option, it is expressed as a length of time or set to `never` in order to turn it off. This setting can, as the previous one, be set to match a `refs` pattern.
- `gc.pruneexpire`: This option tells `git gc` to prune objects older than the value. The default is `2.weeks.ago`, and the value can be expressed as a relative date like `3.months.ago`. To disable the grace period, the value `now` can be used. To set a non-default expiry date only on remote branches, use the following command:

```
$ git config gc./refs/remote/* .reflogexpire never  
$ git config gc./refs/remote/* .reflogexpireunreachable "2  
months"
```

We can also set a date so `git gc` will prune objects sooner:

```
$ git config gc.pruneexpire 3.days.ago
```

Autocorrect

This configuration is useful when you get tired of messages like the following one just because you made a typo on the keyboard:

```
$ git statis  
git: 'statis' is not a git command. See 'git --help'.
```

```
Did you mean this?  
    status
```

By setting the configuration to `help.autocorrect`, you can control how Git will behave when you accidentally send a typo to it. By default, the value is 0 and it means to list the possible options similar to the input (if `statis` is given `status` will be shown). A negative value means to immediately execute the corresponding command. A positive value

means to wait the given number of deciseconds (0.1 sec) before running the command, (so there is some amount of time to cancel it). If several commands can be deduced from the text entered, nothing will happen. Setting the value to half a second gives you some time to cancel a wrong command, as follows:

```
$ git config help.autocorrect 5
$ git statis
WARNING: You called a Git command named 'statis', which does
not exist.
Continuing under the assumption that you meant 'status'
in 0.5 seconds automatically...
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   another-file.txt
#
```

How it works...

Setting the configuration targets will change the way Git behaves. The previous examples describe a few useful methods to get Git to act differently than its default behavior. You should be sure when you are changing a configuration that you completely understand what that configuration does. So, check the Git configuration help page by using `git help config`.

There's more...

There are a lot of configuration targets available in Git. You can run `git help config` and a few pages down all of them are displayed and explained.

Git aliases

An alias is a nice way to configure long and/or complicated Git commands to represent short useful ones. An alias is simply a configuration entry under the alias section. It is usually configured to --global to apply it everywhere.

Getting ready

In this example, we will use the `jgit` repository, which was also used in [Chapter 1, Navigating Git](#), with the `master` branch pointing at `b14a93971837610156e815ae2eee3baaa5b7a44b`. Either use the clone from [Chapter 1, Navigating Git](#), or clone the repository again, as follows:

```
$ git clone https://git.eclipse.org/r/jgit/jgit
$ cd jgit
$ git checkout master && git reset --hard b14a939
```

How to do it...

First, we'll create a few simple aliases, then a couple of more special ones, and finally a couple of aliases using external commands. Instead of writing `git checkout` every time we need to switch branches, we can create an alias of that command and call it `git co`. We can do the same for `git branch`, `git commit`, and `git status` as follows:

```
$ git config --global alias.co checkout
$ git config --global alias.br branch
$ git config --global alias.ci commit
$ git config --global alias.st status
```

Now, try to run `git st` in the `jgit` repository as follows:

```
$ git st
# On branch master
nothing to commit, working directory clean
```

The alias method is also good to create the Git commands you think are missing in Git. One of the common Git aliases is `unstage`, which is used

to move a file out of the staging area, as shown in the following command:

```
$ git config --global alias.unstage 'reset HEAD --'
```

Try to edit the `README.md` file in the root of the `jgit` repository and add it in the root. Now, `git status/git st` should display something like the following:

```
$ git st
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   README.md
#
```

Let's try to unstaged `README.md` and then look at `git st` as follows:

```
$ git unstage README.md
Unstaged changes after reset:
M       README.md

$ git st
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in
working directory)
#
#       modified:   README.md
#
no changes added to commit (use "git add" and/or "git commit -a")
```

A common use case for aliases is to format the history of Git in specific ways. Let's say you want the number of lines added and deleted for each file in the commit displayed along with some common commit data. For this, we can create the following alias so we don't have to type everything each time:

```
$ git config --global alias.ll "log --
pretty=format:\"%C(yellow)%h%Cred%d %Creset%s %Cgreen(%cr)"
```

```
%C(bold blue)<%an>%Creset\" --numstat"
```

Now, we can execute `git ll` in the terminal and get a nice stat output, as shown in the following command:

```
$ git ll
b14a939 (HEAD, master) Prepare 3.3.0-SNAPSHOT builds (8 days
ago) <Matthias Sohn>
6      6      org.eclipse.jgit.ant.test/META-INF/MANIFEST.MF
1      1      org.eclipse.jgit.ant.test/pom.xml
3      3      org.eclipse.jgit.ant/META-INF/MANIFEST.MF
1      1      org.eclipse.jgit.ant/pom.xml
4      4      org.eclipse.jgit.archive/META-INF/MANIFEST.MF
2      2      org.eclipse.jgit.archive/META-INF/SOURCE-
MANIFEST.MF
1      1      org.eclipse.jgit.archive/pom.xml
6      6      org.eclipse.jgit.console/META-INF/MANIFEST.MF
1      1      org.eclipse.jgit.console/pom.xml
12     12     org.eclipse.jgit.http.server/META-
INF/MANIFEST.MF
...
...
```

It is also possible to use an external command instead of a Git command. So, small shell scripts and so on can be embedded. To create an alias with an external command, the alias must start with an exclamation mark !. The examples can be used when resolving conflicts from a rebase or merge. In your `~/.gitconfig` file under [alias], add the following:

```
editconflicted = "!f() {git ls-files --unmerged | cut -f2 |
sort -u ; }; $EDITOR 'f'"
```

This will bring up your configured `$EDITOR` with all the files that are in the conflict state due to the merge/rebase. This quickly allows you to fix the conflicts and get on with the merge/rebase.

In the `jgit` repository, we can create two branches at an earlier point in time and merge these two branches:

```
$ git branch A 03f78fc
$ git branch B 9891497
$ git checkout A
Switched to branch 'A'
```

```
$ git merge B
```

Now, you'll see that this fails to perform the merge, and you can run `git st` to check the status of a lot of files that are in a conflicted state, both modified. To open and edit all the conflicted files, we can now run `git editconflicted`. This brings up `$EDITOR` with the files. If your environment variable isn't set, use `EDITOR=<you-favorite-editor>` `export` to set it.

For this example, we don't actually resolve the conflicts. Just check that the alias works and you're ready for the next alias.

Now that we have solved all the merge conflicts, it is time to add all of those files before we conclude the merge. Luckily, we can create an alias that can help us with that, as follows:

```
addconflicted = "!f() { git ls-files --unmerged | cut -f2 | sort -u ; }; git add 'f'"
```

Now, we can run `git addconflicted`. Later, `git status` will tell us that all the conflicted files are added:

```
$ git st
On branch A
All conflicts fixed but you are still merging.
(use "git commit" to conclude merge)
```

Changes to be committed:

```
modified:   org.eclipse.jgit.console/META-INF/MANIFEST.MF
modified:   org.eclipse.jgit.console/pom.xml
modified:   org.eclipse.jgit.http.server/META-INF/MANIFEST.MF
modified:   org.eclipse.jgit.http.server/pom.xml
modified:   org.eclipse.jgit.http.test/META-INF/MANIFEST.MF
modified:   org.eclipse.jgit.http.test/pom.xml
...
...
```

Now we can conclude the merge with `git commit`:

```
$ git commit
[A 94344ae] Merge branch 'B' into A
```

How it works...

Git simply runs the command the alias is short for. It is very convenient for long Git commands, or Git commands that are hard to remember exactly how to write. Now, all you have to remember is the alias and you can always look in the configuration file for it.

There's more...

Another way to create a kind of Git alias is to make a shell script and save the file with the name `git-<your-alias-name>`. Make the file executable and place it somewhere in your `$PATH`. You can now run that file simply by running `git <your-alias-name>` from the command line.

The refspec exemplified

Though the refspec isn't the first thing that comes to mind when thinking about the Git configuration, it is actually quite close. In a lot of the Git commands the refspec is used, but often implicitly, that is, the refspec is taken from the configuration file. If you don't remember setting a refspec configuration, you are probably right, but if you cloned the repository or added a remote, you'll have a section in `.git/config`, which looks something like the following (this is for the Jgit repository):

```
[remote "origin"]
  url = https://git.eclipse.org/r/jgit/jgit
  fetch = +refs/heads/*:refs/remotes/origin/*
```

The fetch line contains the configured refspec to fetch for this repository.

Getting ready

In this example, we'll be using the `jgit` repository as our server repository, but we have to make a clone of it to a bare repository so we can push it. You can't push to the checked out branch on a non-bare repository as this can overwrite the work area and index.

Create a bare repository from the `jgit` repository and create a new Git repository where we can play with the refspec as follows:

```
$ git clone --bare https://git.eclipse.org/r/jgit/jgit jgit-
bare.git
$ git init refspec-tests
$ cd refspec-tests
$ git remote add origin ../jgit-bare.git
```

We also need to change the branch names on some of the branches to match the example for name spacing; the following will rename the `stable-xxx` branches to `stable/xxx`:

```
$ for br in $(git branch -a | grep "stable-"); do new=$(echo
$br| sed 's/-/\//'); git branch $new $br; done
```

In the previous shell scripting, the \$new and \$br variables aren't placed in double quotes ("") as good practice for shell scripting would otherwise suggest. This is okay as the variables reflect the names of the branches in the repository and branch names cannot contain spaces.

How to do it...

Let us set up our new repository to only fetch the master branch. We do this by changing the fetch line under [remote "origin"] in the configuration file (.git/config), as follows:

```
[remote "origin"]
  url = ../jgit-bare.git
  fetch = +refs/heads/master:refs/remotes/origin/master
```

Now, we will only fetch the master branch and not all the other branches when executing a git fetch, git pull, or a git remote update origin, as follows:

```
$ git pull
remote: Counting objects: 44033, done.
remote: Compressing objects: 100% (6927/6927), done.
remote: Total 44033 (delta 24063), reused 44033 (delta 24063)
Receiving objects: 100% (44033/44033), 9.45 MiB | 5.70 MiB/s,
done.
Resolving deltas: 100% (24063/24063), done.
From ../jgit-bare
 * [new branch]      master      -> origin/master
From ../jgit-bare
 * [new tag]          v0.10.1    -> v0.10.1
 * [new tag]          v0.11.1    -> v0.11.1
 * [new tag]          v0.11.3    -> v0.11.3
...
$ git branch -a
* master
  remotes/origin/master
```

Let's also set up a separate refspec to fetch all the stable/* branches to the local repository as follows:

```
[remote "origin"]
  url = ../jgit-bare.git
  fetch = +refs/heads/master:refs/remotes/origin/master
```

```
fetch =
+refs/heads/stable/*:refs/remotes/origin/stable/*
```

Now, fetch the branches locally, as shown in the following command:

```
$ git fetch
From ../jgit-bare
 * [new branch]      stable/0.10 -> origin/stable/0.10
 * [new branch]      stable/0.11 -> origin/stable/0.11
 * [new branch]      stable/0.12 -> origin/stable/0.12
 * [new branch]      stable/0.7 -> origin/stable/0.7
 * [new branch]      stable/0.8 -> origin/stable/0.8
 * [new branch]      stable/0.9 -> origin/stable/0.9
 * [new branch]      stable/1.0 -> origin/stable/1.0
 * [new branch]      stable/1.1 -> origin/stable/1.1
 * [new branch]      stable/1.2 -> origin/stable/1.2
 * [new branch]      stable/1.3 -> origin/stable/1.3
 * [new branch]      stable/2.0 -> origin/stable/2.0
 * [new branch]      stable/2.1 -> origin/stable/2.1
 * [new branch]      stable/2.2 -> origin/stable/2.2
 * [new branch]      stable/2.3 -> origin/stable/2.3
 * [new branch]      stable/3.0 -> origin/stable/3.0
 * [new branch]      stable/3.1 -> origin/stable/3.1
 * [new branch]      stable/3.2 -> origin/stable/3.2
```

We can also set up push refs that specify where branches are pushed to by default. Let's create a branch called `develop` and create one commit, as shown in the following commands:

```
$ git checkout -b develop
Switched to a new branch 'develop'
$ echo "This is the developer setup, read carefully" > readme-dev.txt
$ git add readme-dev.txt
$ git commit -m "adds readme file for developers"
[develop ccb2f08] adds readme file for developers
1 file changed, 1 insertion(+)
 create mode 100644 readme-dev.txt
```

Now, let's create a push refspec that will send the contents of the `develop` branch to `integration/master` on `origin`:

```
[remote "origin"]
  url = ../jgit-bare.git
  fetch = +refs/heads/master:refs/remotes/origin/master
```

```
fetch = +refs/heads/stable/*:refs/remotes/origin/stable/*
push =
refs/heads/develop:refs/remotes/origin/integration/master
```

Let us push our commit on develop as follows:

```
$ git push
Counting objects: 4, done.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 345 bytes | 0 bytes/s, done.
Total 3 (delta 1), reused 0 (delta 0)
To ../jgit-bare.git
 * [new branch]      develop -> origin/integration/master
```

As the integration/master branch didn't exist on the remote side, it was created for us.

How it works...

The format of the refspec is in the form of <source>:<destination>. For a fetch refspec, this means that <source> is the source on the remote side and <destination> is local. For a push refspec, <source> is local and <destination> is remote. The refspec can be prefixed by a + to indicate that the ref pattern can be updated even though it isn't a fast-forward update. It is not possible to use partial globs in the refspec pattern, as shown in the following line:

```
fetch = +refs/heads/stable*:refs/remotes/origin/stable*
```

But it is possible to use namespacing. That's why we had to rewrite the stable-xxx branches to stable/xxx to fit as a namespace pattern:

```
fetch = +refs/heads/stable/*:refs/remotes/origin/stable/*
```

Chapter 3. Branching, Merging, and Options

In this chapter, we will cover the following recipes:

- Managing your local branches
- Branches with remotes
- Forcing a merge commit
- Using git rerere to merge Git conflicts
- The difference between branches

Introduction

If you are developing a small application in a big corporation as a developer, or you are trying to wrap your head around an open source project from GitHub, you have already been using branches with Git.

Most of the time, you may just be working on a local `develop` or `master` branch and didn't care so much about other branches.

In this chapter, we will show you different branch types and how to work with them.

Managing your local branches

Suppose you are just having your local Git repository, and you have no intentions at the moment to share the code with others; however, you can easily share this knowledge while working with a repository with one or more remotes. Local branches with no remotes work exactly in this fashion. As you can see in the examples, we are cloning a repository, thus we have a remote.

Let's start by creating a few local branches.

Getting ready

Use the following command to clone the `jgit` repository to match:

```
$ git clone https://git.eclipse.org/r/jgit/jgit  
$ cd jgit
```

How to do it...

Perform the following steps to manage your local branches:

1. Whenever you start working on a bug fix or a new feature in your project, you should create a branch. You can do so using the following code:

```
$ git branch newBugFix  
$ git branch  
* master  
  newBugFix
```

2. The `newBugFix` branch points to the current `HEAD` I was on at the time of the creation. You can see the `HEAD` with `git log -1`:

```
$ git log -1 newBugFix --format=format:%H  
25fe20b2dbb20cac8aa43c5ad64494ef8ea64ffc
```

3. If you want to add a description to the branch, you can do it with the `--edit-description` option for the `git branch` command:

```
$ git branch --edit-description newBugFix
```

4. The previous command will open an editor where you can type in a description:

```
Refactoring the Hydro controller  
The hydro controller code is currently horrible needs to be  
refactored.
```

5. Close the editor and the message will be saved.

How it works...

Git stores the information in the local `git config` file; this also means that you cannot push this information to a remote repository.

To retrieve the description for the branch, you can use the `--get` flag for the `git config` command:

```
$ git config --get branch.newBugFix.description  
Refactoring the Hydro controller  
The hydro controller code is currently horrible needs to be  
refactored.
```

This will be beneficial when we automate some tasks in [Chapter 7, Enhancing Your Daily Work with Git Hooks, Aliases, and Scripts](#).

Tip

Remember to perform a checkout of `newBugFix` before you start working on it. This must be done with the Git checkout of `newBugFix`.

The branch information is stored as a file in
`.git/refs/heads/newBugFix`:

```
$ cat .git/refs/heads/newBugFix  
25fe20b2dbb20cac8aa43c5ad64494ef8ea64ffc
```

Note that it is the same commit hash we retrieved with the `git log` command

There's more...

Maybe you want to create specific branches from specific commit hashes. The first thought might be to check out the commit, and then create a branch; however, it is much easier to use the `git branch` command to create the branches without checking out the commits:

1. If you need a branch from a specific commit hash, you can create it with the `git branch` command as follows:

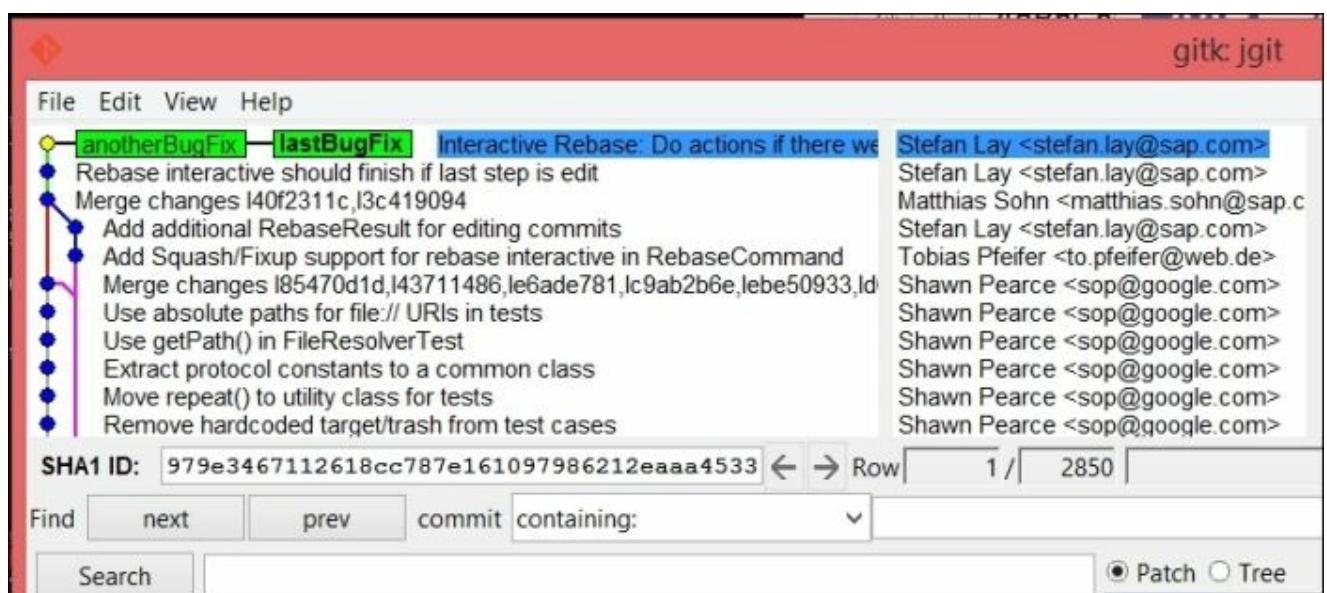
```
$ git branch anotherBugFix 979e346
$ git log -1 anotherBugFix --format=format:%h
979e346
$ git log -1 anotherBugFix --format=format:%H
979e3467112618cc787e161097986212aaaa4533
```

2. As you can see, the abbreviated commit hash is shown when you use `h`, and the full commit hash is shown when you use `H`. You can see the abbreviated commit hash is the same as the one used to create the branch. Most of the time, you want to create and start working on the branch immediately:

```
$ git checkout -b lastBugFix 979e346
Switched to a new branch 'lastBugFix'
```

3. Git switches to the new branch immediately after it creates the branch. Verify with Gitk to see whether the `lastBugFix` branch is checked out and another `BugFix` branch is at the same commit hash:

```
$ gitk
```



- Instead of using Gitk, you can also add `-v` to the `git branch` command or even another `v`.

```
$ git branch -v
  anotherBugFix 979e346 Interactive Rebase: Do actions if
* lastBugFix    979e346 Interactive Rebase: Do actions if
  master        25fe20b Add missing package import for jg
  newBugFix     25fe20b Add missing package import for jg
```

- With `-v`, you can see the abbreviated commit hash for each branch and with `-vv`, you can also see that the `master` branch tracks the `origin/master` branch:

```
$ git branch -vv
  anotherBugFix 979e346 Interactive Rebase: Do actions if e
* lastBugFix    979e346 Interactive Rebase: Do actions if e
  master        25fe20b [origin/master] Add missing package
  newBugFix     25fe20b Add missing package import for g
```

Branches with remotes

At some point, it is very likely that you have cloned somebody's repository. This means you have an associated remote. The remote is usually called origin because it is where the source originated from.

While working with Git and remotes, you will get some benefits from Git.

We can start with `git status` and see what we get while working with the remote.

Getting ready

1. We will start by checking out a local branch that tracks a remote branch:

```
$ git checkout -b remoteBugFix --track origin/stable-3.2
Branch remoteBugFix set up to track remote branch stable-3.2 from origin.
Switched to a new branch 'remoteBugFix'
```

2. The previous command creates and checks out the `remoteBugFix` branch that will track the `origin/stable-3.2` branch. So, for instance, executing `git status` will automatically show how different your branch is from `origin/stable-3.2`, and it will also show whether your branch's `HEAD` can be fast forwarded to the `HEAD` of the remote branch or not.
3. To provide an example of how the previous step works, we need to do some manual work that will simulate this situation. First, we find a commit:

```
$ git log -10 origin/stable-3.2 --oneline
f839d38 Prepare post 3.2.0 builds
699900c JGit v3.2.0.201312181205-r
0ff691c Revert "Fix for core.autocrlf=input resulting in modified fil
1def0a1 Fix for core.autocrlf=input resulting in modified file and un
0ce61ca Canonicalize worktree path in BaseRepositoryBuilder
```

```
if set vi
be7942f Add missing @since tags for new public methods in
Config
ea04d23 Don't use API exception in RebaseTodoLine
3a063a0 Merge "Fix aborting rebase with detached head" into
stable-3.
e90438c Fix aborting rebase with detached head
2e0d178 Add recursive variant of Config.getNames() methods
```

4. The command will list the last 10 commits on the `stable-3.2` branch from the remote origin. The `--oneline` option will show the abbreviated commit hash and the commit subject. For this recipe, we will be using the following commit:

```
$ git reset --hard 2e0d178
HEAD is now at 2e0d178 Add recursive variant of
Config.getNames() m
```

5. This will reset the `remoteBugFix` branch to the `2e0d178` commit hash. We are now ready to continue using the free benefits of Git when we have a remote tracking branch.

We are resetting to a commit that is accessible from the `origin/stable-3.2` remote tracking branch; this is done to simulate that we have performed a Git fetch and new commits were downloaded for the `origin/stable-3.2` branch.

How to do it...

Here, we will try a few commands that assist you when you have a remote tracking branch:

1. Start by executing `git status`:

```
$ git status
On branch remoteBugFix
Your branch is behind 'origin/stable-3.2' by 9 commits, and
can be fast-forwarded.
(use "git pull" to update your local branch)
nothing to commit, working directory clean
```

Git is very descriptive when you have a tracking branch and you use

git status. As you can see from the message, you can use git pull to update your local branch, which we will try in the next example. Now, we will just perform the merge:

Tip

The git pull command is just a git fetch command and then a git merge command with the remote tracking branch.

```
$ git merge origin/stable-3.2
Updating 2e0d178..f839d38
Fast-forward
  .../org/eclipse/jgit/api/RebaseCommandTest.java      | 213
  ++++++
  .../src/org/eclipse/jgit/api/RebaseCommand.java      |  31
  +-+
  .../jgit/errors/IllegalTodoFileModification.java    |  59
  ++++++
  .../eclipse/jgit/lib/BaseRepositoryBuilder.java     |   2
  +-+
  .../src/org/eclipse/jgit/lib/Config.java            |   2
+.../src/org/eclipse/jgit/lib/RebaseTodoLine.java     |  16
  +-+
  6 files changed, 266 insertions(+), 57 deletions(-)
  create mode 100644
  org.eclipse.jgit/src/org/eclipse/jgit/errors/Ille
```

2. From the output, you can see it is a fast-forward merge, as Git predicted in the output of git status.

There's more...

You can also add a remote to an existing branch, which is very handy when you realize that you actually wanted a remote tracking branch but forgot to add the tracking information while creating the branch:

1. Start by creating a local branch at the 2e0d17 commit:

```
$ git checkout -b remoteBugFix2 2e0d17
```

```
Switched to a new branch 'remoteBugFix2'
```

2. The remoteBugFix2 branch is just a local branch at the moment with no tracking information; to set the tracking branch, we need to use -

-set-upstream-to or -u as a flag to the git branch command:

```
$ git branch --set-upstream-to origin/stable-3.2
Branch remoteBugFix2 set up to track remote branch stable-3.2 from origin.
```

- As you can see from the Git output, we are now tracking the stable-3.2 branch from the origin:

```
$ git status
On branch remoteBugFix2
Your branch is behind 'origin/stable-3.2' by 9 commits, and
can be fast-forwarded.

(use "git pull" to update your local branch)

nothing to commit, working directory clean
```

- You can see from the Git output that you are nine commits ahead, and you can use git pull to update the branch. Remember that a git pull command is just a git fetch command, and then a git merge command with the upstream branch, which we also call the remote tracking branch:

```
$ git pull
remote: Counting objects: 1657, done
remote: Finding sources: 100% (102/102)
remote: Total 102 (delta 32), reused 98 (delta 32)
Receiving objects: 100% (102/102), 65.44 KiB | 0 bytes/s,
done.
Resolving deltas: 100% (32/32), completed with 19 local
objects.
From https://git.eclipse.org/r/jgit/jgit
 25fe20b..50a830f  master      -> origin/master
First, rewinding head to replay your work on top of it...
Fast-forwarded remoteBugFix2 to
f839d383e6fbnda26729db7fd57fc917fa47db44.
```

- From the output, you can see the branch has been fast forwarded to the f839d383e6fbnda26729db7fd57fc917fa47db44 commit hash, which is equivalent to origin/stable-3.2. You can verify this with git log:

```
$ git log -1 origin/stable-3.2 --format=format:%H
f839d383e6fbnda26729db7fd57fc917fa47db44
```

Forcing a merge commit

If you are reading this book, you might have seen a lot of basic examples of software delivery chains and branching models. It is very likely that you have been trying to use different strategies and found that none of them completely supports your scenario, which is perfectly fine as long as the tool can support your specific workflow.

Git supports almost any workflow. I have often encountered a situation that requires a merge commit while merging a feature, even though it can be done with a fast-forward merge. Those who requested it often use it to indicate that you have actually merged in a feature and want to store the information in the repository.

Tip

Git has fast and easy access to all the commit messages, so the repository should be used as a journal, not just a *backup* of the source code.

Getting ready

Start by checking out a local branch `remoteOldbugFix` that tracks `origin/stable-3.1`:

```
$ git checkout -b remoteOldBugFix --track origin/stable-3.1
Branch remoteOldBugFix set up to track remote branch stable-3.1
from Switched to a new branch 'remoteOldBugFix'
```

How to do it...

The following steps will show you how to force a merge commit:

1. To force a merge commit, you need to use the `--no-ff` flag, `no-ff` is no fast forward. We will also use `--quiet` for minimizing the output and `--edit` to allow us to edit the commit message. Unless you have a merge conflict, Git will create the merge commit for you automatically:

```
$ git merge origin/stable-3.2 --no-ff --edit --quiet
Auto-merging
org.eclipse.jgit.test/tst/org/eclipse/jgit/test/resources/S
ampleDat
Removing
org.eclipse.jgit.test/tst/org/eclipse/jgit/internal/storage
/file/GCTe
Auto-merging
org.eclipse.jgit.packaging/org.eclipse.jgit.target/jgit-
4.3.target
```

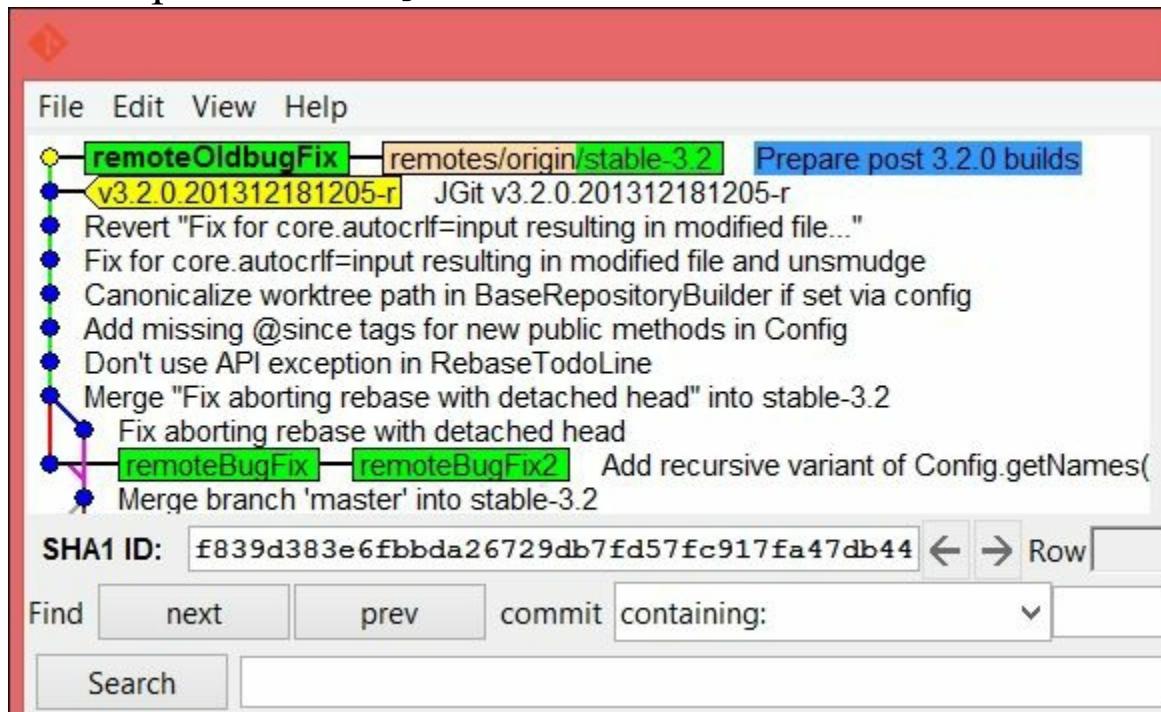
2. The commit message editor will open, and you can write a commit message. Closing the editor creates the merge commit and we are done.
3. To verify this, you can reset back to `origin/stable-3.1` and perform the merge without the `--no-ff` flag:

```
$ git reset --hard remotes/origin/stable-3.1
HEAD is now at da6e87b Prepare post 3.1.0 builds
```

4. Now, perform the merge with the following command:

```
$ git merge origin/stable-3.2 --quiet
```

5. You can see the difference using Gitk. The following screenshot shows the fast forward merge, as you can see our `remoteOldBugFix` branch points to `origin/stable-3.2`:



6. The following screenshot shows the merge commit we forced Git to create. My branch `remoteOldBugFix` is ahead of `remotes/origin/stable-3.2`, and I performed the commit:

The screenshot shows a Git commit history window. At the top, it says "File Edit View Help". Below that, there's a title bar with "remoteOldbugFix Merging stable-3.2". The commit list starts with "remotes/origin/stable-3.2 Prepare post 3.2.0 builds" (SHA1: 93726404cee93dcc6461e45c48917fef54cddee58). This is followed by several commits from "v3.2.0.201312181205-r" (JGit v3.2.0.201312181205-r), including "Revert 'Fix for core.autocrlf=input resulting in modified file...'" and "Merge 'Fix aborting rebase with detached head' into stable-3.2". The final commit shown is "remoteBugFix remoteBugFix2 Add recursive variant of Config.g" (SHA1: 93726404cee93dcc6461e45c48917fef54cddee58). At the bottom, there are buttons for "Find", "next", "prev", "commit", "containing:", and a "Search" field.

There's more...

Although most branching scenarios expect you to completely merge branches, there are situations when while working in a real environment, you only need to merge specific pieces of one branch into another branch. Using the `--no-commit` option, Git will make the merge and stop before committing, allowing you to modify and add files to the merge commit before committing.

As an example, I have been working with projects where versions of strings have been updated in the `feature` branch but not in the `master` branch. So, an automatic merge into `master` would replace the current version string used on the `master` branch, which in my case was not the intention:

1. Start by checking out a local `remotePartlyMerge` branch that tracks `origin/stable-2.3`:

```
$ git checkout -b remotePartlyMerge --track origin/stable-2.3
Branch remotePartlyMerge set up to track remote branch stable-2.3 from origin.
Switched to a new branch 'remotePartlyMerge'
```

2. Then, to create the merge and allow you to decide what will be part of the commit, you can use --no-commit:

```
$ git merge origin/stable-3.1 --no-ff --no-commit
a lot of output...
Automatic merge went well; stopped before committing as requested
```

3. Again, Git is very informative; you can see from the output that everything went well and Git stopped before committing as requested. To continue, let's pretend we didn't want the org.eclipse.jgit.test directory to be part of the merge commit. To achieve this, we reset the directory using the git reset <path> command:

```
$ git reset ./org.eclipse.jgit.test
Unstaged changes after reset:
M      org.eclipse.jgit.test/.gitignore
A lot of output
M
org.eclipse.jgit.test/tst/org/eclipse/jgit/util/io/AutoCRLF
OutputStreamTest.java
```

4. You can see from the output that you have unstaged changes after the reset; this is exactly what we want. You can check which unstaged changes you have by running git status. Now, we will just finish the merge:

```
$ git commit -m "Merging stable-3.1 without
org.eclipse.jgit.test"
[remotePartlyMerge 396f32a] Merging stable-3.1 without
```

5. The merge commit is complete. If you run a git status command now, you will still have the unstaged changes in your work area. To verify whether the result is as expected, we can use diff for this with git diff to show that the file is as it is on the origin/stable-2.3 branch:

```
$ git diff origin/stable-2.3 ./org.eclipse.jgit.test
```

6. There is no output from `diff`; this is the expected result. We are telling the `diff` command to diff our current `HEAD` commit and branch `origin/stable-2.3`, and we only care about the diffs in `./org.eclipse.jgit.test`:

Tip

If you don't specify `HEAD`, you will diff with your current WA, and the `diff` command will have a lot of output as you have unstaged changes.

Using git rerere to merge known conflicts

While working on a feature branch, you probably like to merge daily or maybe more often, but often when you work on long-living feature branches, you end up in a situation where you have the same conflict occurring repeatedly.

Here, you can use `git rerere` which stands for *reuse recorded resolution*. Git `rerere` is not enabled by default but can be enabled with the following command:

```
$ git config rerere.enabled true
```

Tip

You can configure it globally by adding `--global` to the `git config` command.

How to do it...

Perform the following steps to merge the known conflicts:

1. In the `jgit` repository folder, start by checking out a branch that tracks `origin/stable-2.2`:

```
git checkout -b rerereExample --track origin/stable-2.2
```

2. Now, change the maven-compiler-plugin version to something personalized such as `2.5.2` like this is in line 211. If you run `git diff`, you should get a result very similar to the following:

```
$ git diff
diff --git a/pom.xml b/pom.xml
index 085e00f..d5aec17 100644
--- a/pom.xml
+++ b/pom.xml
@@ -208,7 +208,7 @@
```

`<plugin>`

```
<artifactId>maven-compiler-plugin</artifactId>
-
<version>2.5.1</version>
+
<version>2.5.2</version>
</plugin>

<plugin>
```

3. Now add the file and create a commit:

```
$ git add pom.xml
$ git commit -m "Update maven-compiler-plugin to 2.5.2"
[rerereExample d474848] Update maven-compiler-plugin to
2.5.2
 1 file changed, 1 insertion(+), 1 deletion(-)
```

4. Store your current commit in a backup branch named rerereExample2:

```
$ git branch rerereExample2
```

Here, git branch rerereExample2 is just storing the current commit as a branch, as we need to use that for rerere example number 2.

5. Now, we need to perform the first merge that will fail on auto merge. Then, we can solve that. After solving it, we can reuse the merge resolution to solve the same problem in the future:

```
$ git merge --no-ff v3.0.2.201309041250-rc2
A lot of output ...
Automatic merge failed; fix conflicts and then commit the
result.
```

6. As we have git rerere enabled, we can use git rerere status to see which files or paths will be recorded:

```
$ git rerere status
pom.xml
```

7. Edit the pom.xml file and solve the merge conflict so that you can get the diff output shown as follows. You have to remove the line with 2.5.1 and the merge markers:

Tip

Merge markers are lines that begin with <<<<<, >>>>>, or =====; these lines indicate the points where Git could not perform an auto

merge.

```
$ git diff v3.0.2.201309041250-rc2 pom.xml
diff --git a/pom.xml b/pom.xml
index 60cb0c8..faa7618 100644
--- a/pom.xml
+++ b/pom.xml
@@ -226,7 +226,7 @@

<plugin>
    <artifactId>maven-compiler-plugin</artifactId>
-
    <version>3.1</version>
+
    <version>2.5.2</version>
</plugin>

<plugin>
```

8. Mark the merge as complete by adding `pom.xml` to the staging area and run `git commit` to finish the merge:

```
$ git commit
Recorded resolution for 'pom.xml'.
[rerereExample 9b8725f] Merge tag 'v3.0.2.201309041250-rc2'
into rerereExample
```

9. Note the recorded resolution for the `pom.xml` output from Git; this will not be here without enabling `git rerere`. Git has recorded this resolution to this particular merge conflict and will also record how to solve this. Now, we can try and rebase the change to another branch.
10. Start by checking out the `rerereExample2` branch from our repository:

```
$ git checkout rerereExample2
Switched to branch 'rerereExample2'
```

11. Try to rebase your change on top of the `origin/stable-3.2` branch:

```
$ git rebase origin/stable-3.2
First, rewinding head to replay your work on top of it...
Applying: Update maven-compiler-plugin to 2.5.2
Using index info to reconstruct a base tree...
M      pom.xml
Falling back to patching base and 3-way merge...
Auto-merging pom.xml
```

```
CONFLICT (content): Merge conflict in pom.xml
Resolved 'pom.xml' using previous resolution.
Failed to merge in the changes.
Patch failed at 0001 Update maven-compiler-plugin to 2.5.2
The copy of the patch that failed is found in:
  c:/Users/Rasmus/repos/jgit/.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".

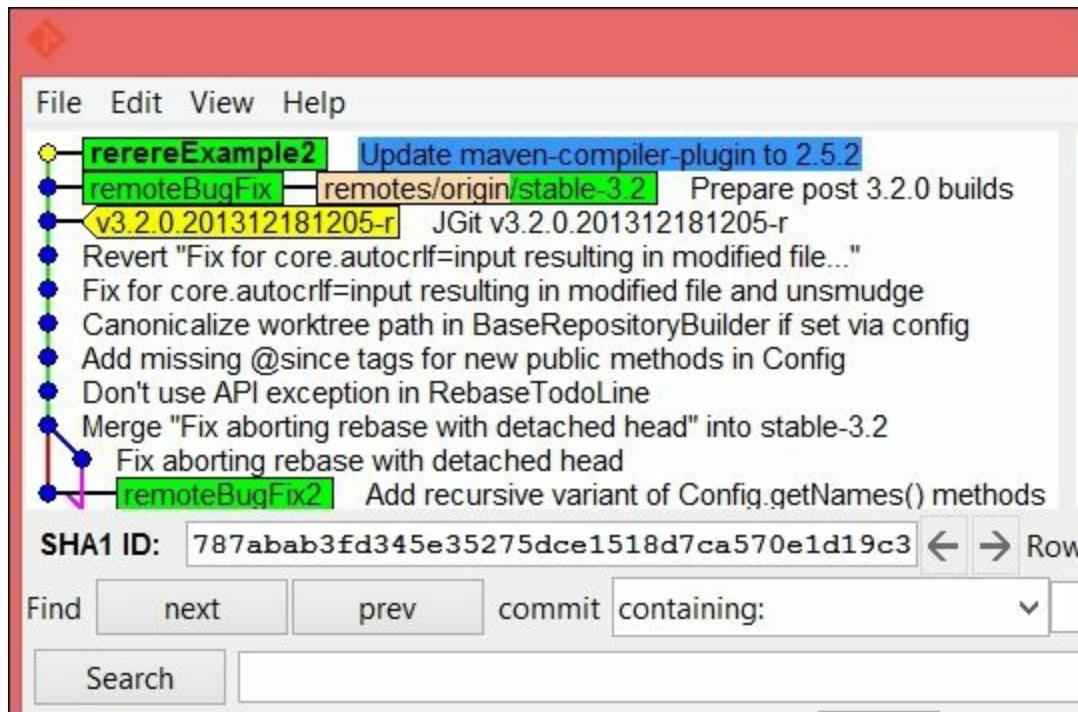
12. You should notice the following output:

```
CONFLICT (content): Merge conflict in pom.xml
Resolved 'pom.xml' using previous resolution
```

13. As the merge conflict is the same in pom.xml, Git can solve the conflict in the file for you. This is very clear when you open the file and see there are no merge markers, as the resolution Git had recorded has been applied. Finish the merge by adding pom.xml and continue the rebase:

```
$ git add pom.xml
$ git rebase --continue
Applying: Update maven-compiler-plugin to 2.5.2
```

14. Start Gitk to see that the commit has been rebased on top of the origin/stable-3.2 branch:



You can try the same scenario with merging and it would merge the file automatically for you.

There's more...

When you merge different branches often and you are not sure which branch a specific error fix is a part of, it is actually quite easy to find out.

1. You need to find a commit you are interested in getting this information for. Then, use the --contains flag for the git branch command:

```
$ git branch --contains 06ddde1
  anotherBugFix
  lastBugFix
  master
  newBugFix
  remoteBugFix
  remoteBugFix2
  remoteOldbugFix
* rerereExample2
```

2. The previous command lists all the branches that have the specific commit. If you leave out the <commit> option, Git will check HEAD. So, for instance, checking out the `rerereExample2` branch and executing the following command, you will see the commit is present only on that branch:

```
$ git checkout rerereExample2
Switched to branch 'rerereExample2'
$ git branch -a --contains
* rerereExample2
```

Tip

The `-a` option indicates that you wish to check all the remote branches as well. If you leave this out, it will check only local branches.

However, as you can see, our commit is not on any remote branch as the commit has just been created locally and has not been pushed to any remotes yet.

Tip

You can use tags, branch names, or commit hashes while using the `git branch -a --contains` command.

3. Let's try to see the branches where the `v2.3.0.201302130906` tag is present:

```
$ git branch -a --contains v2.3.0.201302130906
anotherBugFix
lastBugFix
master
newBugFix
remoteBugFix
remoteBugFix2
remoteOldbugFix
remotePartlyMerge
* rerereExample2
remotes/origin/HEAD -> origin/master
remotes/origin/master
remotes/origin/stable-2.3
remotes/origin/stable-3.0
```

`remotes/origin/stable-3.1`

`remotes/origin/stable-3.2`

That tag is in quite a lot of branches.

The difference between branches

Checking the difference between branches can show valuable information before merging.

A regular Git diff between two branches will show you all the information, but it can be rather exhausting to sit and look at; maybe you are only interested in one file. Thus, you don't need the long unified diff.

Getting ready

To start with, we decide on two branches, tags, or commits we want to see the diff between. Then, to list files that have changed between these branches, you can use the `--name-only` flag.

How to do it...

Perform the following steps to see the difference between the branches:

1. Diff the `origin/stable-3.1` with `origin/stable-3.2` branch:

```
$ git diff --name-only origin/stable-3.1 origin/stable-3.2
org.eclipse.jgit/src/org/eclipse/jgit/transport/org.eclipse
.jgit/src/org/eclipse/jgit/transport/BasePackFetch
More output..
```

2. We are building the command in this pattern, that is, `git diff [options] <commit> <commit> <path>`. Then, we can diff what we care about while looking into the differences between branches. This is very useful if you are responsible for a subset of the source code, and you wish to diff that area only.
3. Let's try the same diff between branches, but this time we will diff the entire branches, not just a subdirectory; however, we only want to show the deleted or added files between the branches. This is done by using the `--diff-filter=DA` and `--name-status` options. The `--name-status` option will only show the filenames and the type of change. The `--diff-filter=DA` option will only show the deleted

and added files:

```
$ git diff --name-status --diff-filter=DA origin/stable-3.1  
origin/stable-3.2  
D org.eclipse.jgit.junit/src/org/eclipse/jgit/junit/Sam  
A org.eclipse.jgit.packaging/org.eclipse.jgit.target/jg  
More output..
```

4. This shows the files that have been added and deleted while moving from `origin/stable-3.1` to `origin/stable-3.2`:
5. If we switch the branches around like in the following command, we will get the opposite result.

```
$ git diff --name-status --diff-filter=DA origin/stable-3.2  
origin/stable-3.1  
A org.eclipse.jgit.junit/src/org/eclipse/jgit/junit/Sam  
D org.eclipse.jgit.packaging/org.eclipse.jgit.target/jg  
More output..
```

Note that the indication letters `A` and `D` switched places because now we want to know what happens if we move from `origin/stable-3.2` to `origin/stable-3.1`.

There's more...

There are more options in the help files for Git. Just run `git merge --help` or `git branch --help` to see what other options you have. The option I have used and the examples shown are all examples that have given me the edge while working with Git as a release manager and Git mentor at Nokia.

Chapter 4. Rebase Regularly and Interactively, and Other Use Cases

In this chapter, we will cover the following topics:

- Rebasing commits to another branch
- Continuing a rebase with merge conflicts
- Rebasing selective commits interactively
- Squashing commits using an interactive rebase
- Changing the author of commits using a rebase
- Auto-squashing commits

Introduction

Rebase is an incredibly strong feature of Git. Hopefully, you have used it before; if not, you might have heard about it. Rebasing is exactly what the word implies. So, if you have a certain commit A that is based on commit B, then rebasing A to C would result in commit A being based on commit C.

As you will see in the different examples in this chapter, it is not always as simple as that.

Rebasing commits to another branch

To start with, we are going to perform a very simple rebase where we will introduce a new file, commit this file, make a change to it, and then commit it again so that we end up with 2 new commits.

Getting ready

Before we start, we need a repository to work in. You can use a previous clone of `jgit`, but to get a close to identical output from the example, you can clone the `jgit` repository.

The `jgit` repository can be cloned as follows:

```
$ git clone https://git.eclipse.org/r/jgit/jgit chapter4  
$ cd chapter4
```

How to do it...

We start by creating a local branch and then make two commits by performing the following steps; these are the commits that we want to rebase onto another branch:

1. Check out a new branch, `rebaseExample`, that tracks `origin/stable-3.1`:

```
$ git checkout -b rebaseExample --track origin/stable-3.1  
Branch rebaseExample set up to track remote branch stable-  
3.1 from origin.  
Switched to a new branch 'rebaseExample'
```

2. Make two commits on the `rebaseExample` branch as follows:

```
$ echo "My Fishtank  
  
Gravel, water, plants  
Fish, pump, skeleton" > fishtank.txt  
$ git add fishtank.txt  
$ git commit -m "My brand new fishtank"
```

```
[rebaseExample 4b2c2ec] My brand new fishtank
 1 file changed, 4 insertions(+)
 create mode 100644 fishtank.txt
$ echo "mosquitos" >> fishtank.txt
$ git add fishtank.txt
$ git commit -m "Feeding my fish"
[rebaseExample 2132d88] Feeding my fish
 1 file changed, 1 insertion(+)
```

3. Then, we rebase the change on top of the origin/stable-3.2 branch instead.

```
$ git rebase origin/stable-3.2
First, rewinding head to replay your work on top of it...
Applying: My brand new fishtank
Applying: Feed the fish
```

How it works

When you execute `git rebase`, Git starts by finding the common ancestor of the current `HEAD` branch and the branch you want to rebase to. When Git finds `merge-base`, it will find the commits that are not available on the branch you are rebasing onto. Git will simply try to apply those commits one by one.

Continuing a rebase with merge conflicts

When you rebase a commit or a branch on top of a different `HEAD`, you will eventually see a conflict.

If there is a conflict, you will be asked to solve the merge conflict and continue with the rebase using `git rebase --continue`.

How to do it

We will be creating a commit that adds the same `fishtank.txt` file on top of the `origin/stable-3.1` branch; then, we will try to rebase this on top of the `rebaseExample` branch we created in the *Rebasing commits to another branch* section:

1. Check out a branch named `rebaseExample2` that tracks `origin/stable-3.1`:

```
$ git checkout -b rebaseExample2 --track origin/stable-3.1
Checking out files: 100% (212/212), done.
Branch rebaseExample2 set up to track remote branch stable-3.1 from origin.
Switched to a new branch 'rebaseExample2'
```

2. Make a commit on the branch.

```
$ echo "My Fishtank

Pirateship, Oister shell
Coconut shell
">>fishtank.txt
$ git add fishtank.txt
$ git commit -m "My brand new fishtank"
[rebaseExample2 39811d6] My brand new fishtank
 1 file changed, 5 insertions(+)
 create mode 100644 fishtank.txt
```

3. Try to rebase the branch on top of the `rebaseExample` branch.

```
$ git rebase rebaseExample
```

```

First, rewinding head to replay your work on top of it...
Applying: My brand new fishtank
Using index info to reconstruct a base tree...
<stdin>:12: new blank line at EOF.
+
warning: 1 line adds whitespace errors.
Falling back to patching base and 3-way merge...
Auto-merging fishtank.txt
CONFLICT (add/add): Merge conflict in fishtank.txt
Failed to merge in the changes.
Patch failed at 0001 My brand new fishtank
The copy of the patch that failed is found in:
  c:/Users/Rasmus/repos/chapter4/.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".

- As predicted, we have a conflict. Solve the conflict in your preferred editor. Then, add the file to the index using `git add` and continue with the rebase.

```

$ git add fishtank.txt
$ git rebase --continue
Applying: My brand new fishtank

```

- We can now check with `gitk` to see whether our change is rebased on top of the `rebaseExample` branch, as shown in the following screenshot:



How it works

As we learned from the first example, Git will apply the commits that are not available on the branch you are rebasing to. In our example, it is only our commit, as we made it, that is available on the `rebaseExample2` branch.

There's more...

You might have noticed in the output of the failing rebase that you have two extra options for the commit.

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`.

The first extra option we have is to totally ignore this patch by skipping it; you can do this using `git rebase --skip`. In our example, this will cause our branch to be fast-forwarded to the `rebaseExample` branch. So, both our branches will point to the same commit hash.

The second option is to abort the rebasing. If we choose to do this, then we will go back to the branch as it was prior to starting the rebase. This can be done using `git rebase --abort`.

Rebasing selective commits interactively

When you are working on a new feature and have branched from an old release into a feature branch, you might want to rebase this branch onto the latest release. When looking into the list of commits on the feature branch, you realize that some of the commits are not suitable for the new release. So, when you want to rebase the branch onto a new release, you will need to remove some commits. This can be achieved with interactive rebasing, where Git gives you the option to pick the commits you wish to rebase.

Getting ready

To get started with this example, you need to check the previously created branch, `rebaseExample`; if you don't have this branch, follow the steps from the *Rebasing a few commits* section and use the following command:

```
$ git checkout rebaseExample
Switched to branch 'rebaseExample'
Your branch is ahead of 'origin/stable-3.1' by 109 commits.
  (use "git push" to publish your local commits)
```

Notice that because we are tracking `origin/stable-3.1`, the Git checkout will tell us how far ahead we are in comparison with that branch.

How to do it

We will try to rebase our current branch, `rebaseExample`, on top of the `origin/stable-3.1` branch by performing the following steps.

Remember that Git will apply the commits that are not available on the branch we are rebasing to; so, in this case, there will be a lot of commits:

1. Rebase the branch onto `origin/stable-3.1` by using the following command:

```
$ git rebase --interactive origin/stable-3.1
```

2. What you will see now is a list of all the commits you will be rebasing onto the `origin/stable-3.1` branch. These commits are all the commits between the `origin/stable-3.1` and `rebaseExample` branches. The commits will be applied from top to bottom, so the commits will be listed in reverse order, at least compared to what you would normally see in Git. This actually makes good sense. The commits have the keyword `pick` to the left and then the abbreviated commit hash, and finally the title of the commit subject.

If you scroll down to the bottom, you will see a list like the following one:

```
pick 43405e6 My brand new fishtank
pick 08d0906 Feed the fish
# Rebase da6e87b..08d0906 onto da6e87b
#
# Commands:
#   p, pick = use commit
#   r, reword = use commit, but edit the commit message
#   e, edit = use commit, but stop for amending
#   s, squash = use commit, but meld into previous commit
#   f, fixup = like "squash", but discard this commit's log
message
#   x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top
to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be
aborted.
#
# Note that empty commits are commented out
```

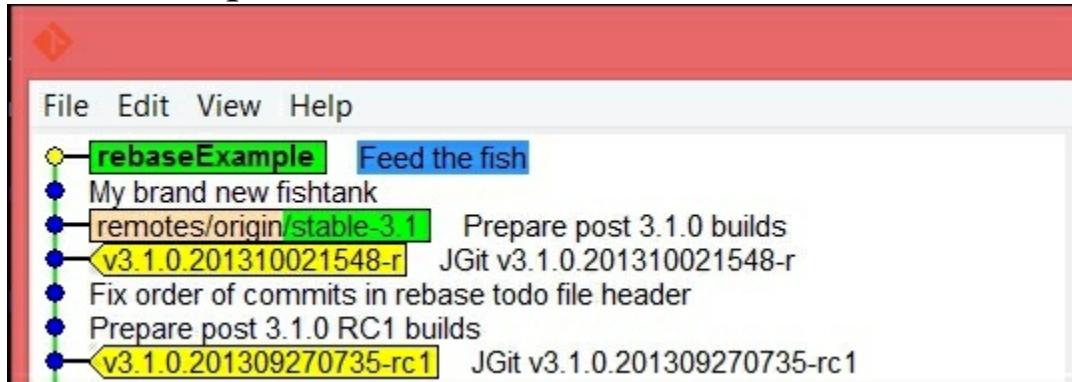
So, if we only want our fishtank commits to be based on top of the `origin/stable-3.1` branch, we should remove all the commits except for our two commits.

3. Remove all the lines except for the two commits at the bottom; for now, leave `pick` as the keyword. Save the file and close the editor,

and you will get the following message from Git:

Successfully rebased and updated refs/heads/rebaseExample.

4. Now, with `gitk`, try to check whether we accomplished what we predicted. The next screenshot shows our two fishtank commits on top of the `origin/stable-3.1` branch. The following screenshot is what we expected:

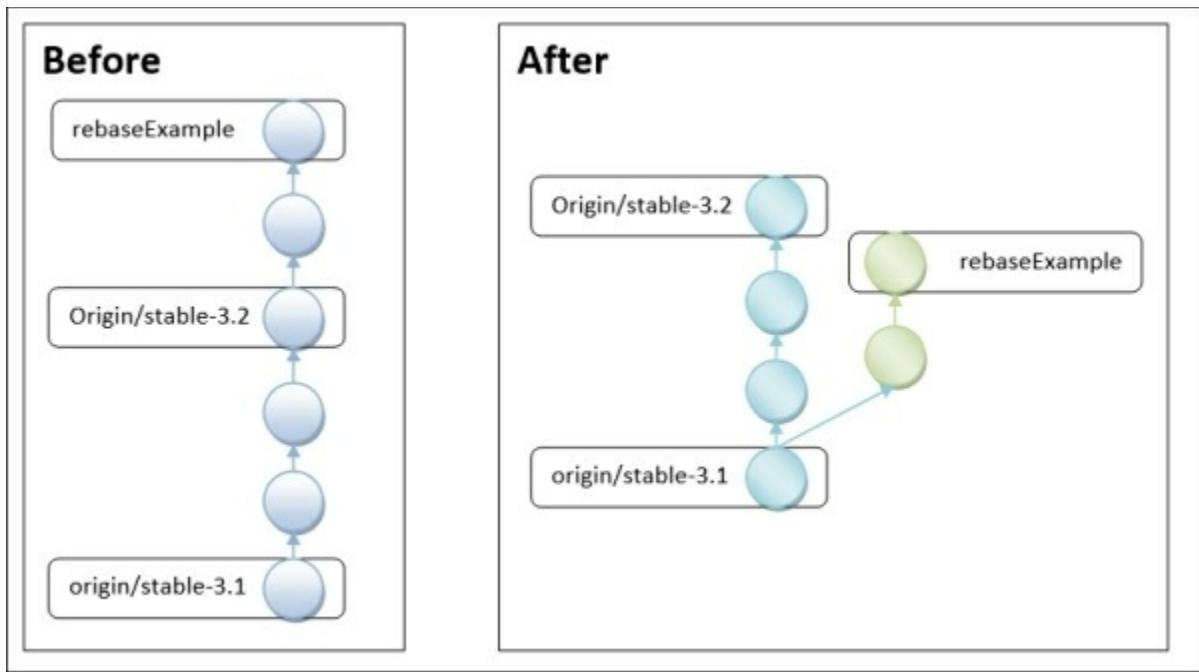


There's more...

The same thing could actually be achieved with a single small Git command. We have been rebasing commits from the `origin/stable-3.2` branch to the `rebaseExample` branch onto the `origin/stable-3.2` branch. This can also be achieved in the following manner:

```
$ git rebase --onto origin/stable-3.1 origin/stable-3.2  
rebaseExample  
First, rewinding head to replay your work on top of it...  
Applying: My brand new fishtank  
Applying: Feed the fish
```

The `--onto origin/stable-3.2` flag tells Git to rebase onto `origin/stable-3.2`, and it has to be from `origin/stable-3.1` to the `rebaseExample` branch. So, we end up having `rebaseExample` branch to the branch of the `origin/stable-3.1` and the like. The next diagram shows before the rebase example where we have our two commits on top of `origin/stable-3.2`, then after the rebase where our commits are on top of `origin/stable-3.1` as we wanted:



Squashing commits using an interactive rebase

When I work on a local branch, I prefer to commit in small increments with a few comments on what I did in the commits; however, as these commits do not build or pass any test requirements, I cannot submit them for review and verification one by one. I have to merge them in my branch, and still, cherry picking my fix would require me to cherry-pick twice the number of commits, which is not very handy.

What we can do is rebase and squash the commits into a single commit or at least a smaller number of commits.

Getting ready

To get started with this example, we need a new branch, namely `rebaseExample3`, that tracks `origin/stable-3.1`. Create the branch with the following command:

```
$ git checkout -b rebaseExample3 --track origin/stable-3.1
Branch rebaseExample3 set up to track remote branch stable-3.1
from origin.
Switched to a new branch 'rebaseExample3'
```

How to do it...

To really showcase this feature of Git, we will start by being six commits ahead of the `origin/stable-3.1` branch. This is to simulate the fact that we have just created six commits on top of the `rebaseExample3` branch; to do this, perform the following steps:

1. Find a commit that is between `origin/stable-3.1` and `origin/stable-3.2` and list the commits in reverse order. If you don't list them in reverse order, you can scroll down to the bottom of the output and find the commit we will use, as shown in the following snippet:

```
$ git log origin/stable-3.1..origin/stable-3.2 --oneline --reverse
8a51c44 Do not close ArchiveOutputStream on error
3467e86 Revert "Close unfinished archive entries on error"
f045a68 Added the git-describe implementation
0be59ab Merge "Added the git-describe implementation"
fdc80f7 Merge branch 'stable-3.1'
7995d87 Prepare 3.2.0-SNAPSHOT builds
5218f7b Propagate IOException where possible when getting refs.
```

2. Reset the `rebaseExample3` branch to the `5218f7b` commit; this will simulate that we have six commits on top of the `origin/stable-3.1` branch. This can be tested by running the status of Git as follows:

```
$ git reset --hard 5218f7b
HEAD is now at 5218f7b Propagate IOException where possible
when getting refs.
$ git status
On branch rebaseExample3
Your branch is ahead of 'origin/stable-3.1' by 6 commits.
  (use "git push" to publish your local commits)

nothing to commit, working directory clean
```

3. Now we have these six commits on top of the `origin/stable-3.1` branch, and we want to squash these commits into two different commits instead of six commits. This can be done by simply running `git rebase --interactive`. Notice that we are not specifying which branch we want to rebase to since we have already set up a tracking branch when we created the branch using `--track`. To continue, let's execute the `rebase` command as follows:

```
$ git rebase --interactive
pick 8a51c44 Do not close ArchiveOutputStream on error
pick f045a68 Added the git-describe implementation
pick 7995d87 Prepare 3.2.0-SNAPSHOT builds
pick 5218f7b Propagate IOException where possible when
getting refs.
```

4. The editor will open, and you will see four commits and not six as you would expect. This is because the rebase in general refuses to take merge commits as part of the rebase scenario. You can use the `--preserve-merges` flag. As per the Help section of Git, this is not

recommended.

Note

According to the Help section in Git `--preserve-merges` instead of ignoring merges, tries to recreate them.

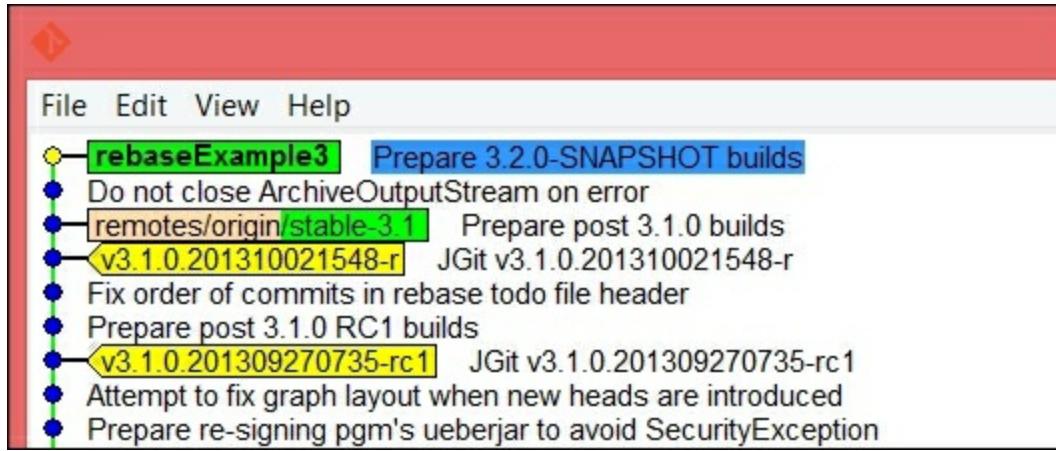
The `--preserve-merges` flag uses the `--interactive` machinery internally, but combining it with the `--interactive` option explicitly is generally not a good idea unless you know what you are doing (see the BUGS in the following snippet).

5. Edit the file so it looks as follows:

```
pick 8a51c44 Do not close ArchiveOutputStream on error
squash f045a68 Added the git-describe implementation
pick 7995d87 Prepare 3.2.0-SNAPSHOT builds
squash 5218f7b Propagate IOException where possible when
getting refs.
```

6. Remember that commits are listed in reverse order as compared to the Git log. So, while squashing, we squash up into the commits we have marked with the pick. When you close the editor, Git will start the rebase from top to bottom. First, apply `8a51c44` and then squash `f045a68` into the commit `8a51c44`. This will open the commit message editor that contains both the commit messages. You can edit the commit messages, but for now, let us just close the editor to finish with the rebase and the squashing of these two commits. The editor will open one more time to complete the squashing of `5218f7b` into `7995d87`. Use `gitk` to verify the result.

The following screenshot is as expected; now, we only have two commits on top of the `origin/stable3-1` branch:



7. If you check the commit message of the `HEAD` commit, you will see that it has the information of two commits, as shown in the following command. This is because we decided not to change the commit message when we made the change:

```
$ git log -1
commit 9c96a651ff881c7d7c5a3974fa7a19a9c264d0a0
Author: Matthias Sohn <matthias.sohn@sap.com>
Date:   Thu Oct 3 17:40:22 2013 +0200

    Prepare 3.2.0-SNAPSHOT builds

    Change-Id: Iac6cf7a5bb6146ee3fe38abe8020fc3fc4217584
    Signed-off-by: Matthias Sohn <matthias.sohn@sap.com>

    Propagate IOException where possible when getting refs.

    Currently, Repository.getAllRefs() and
    Repository.getTags() silently
        ignores an IOException and instead returns an empty
    map. Repository
        is a public API and as such cannot be changed until the
    next major
        revision change. Where possible, update the internal
    jgit APIs to
        use the RefDatabase directly, since it propagates the
    error.

    Change-Id: I4e4537d8bd0fa772f388262684c5c4ca1929dc4c
```

There's more...

Now we have squashed two commits, but we could have used other keywords when editing the rebase's to-do list.

We will try the fixup functionality, which works like the squash functionality, by performing the following steps; the exception is that Git will select the commit message of the commits with the `pick` keyword:

1. Start by resetting back to our starting point.

```
$ git reset --hard 5218f7b
HEAD is now at 5218f7b Propagate IOException where possible
when getting refs.
$ git status
On branch rebaseExample3
Your branch is ahead of 'origin/stable-3.1' by 6 commits.
  (use "git push" to publish your local commits)

nothing to commit, working directory clean
```

2. As you can see, we are back at the starting point, that is, we're six commits ahead of the `origin/stable-3.1` branch. Now, we can try the fixup functionality. Start the interactive rebase and change the file according to the following output. Notice that you can use `f` instead of `fixup`.

```
$ git rebase --interactive
pick 8a51c44 Do not close ArchiveOutputStream on error
f f045a68 Added the git-describe implementation
pick 7995d87 Prepare 3.2.0-SNAPSHOT builds
f 5218f7b Propagate IOException where possible when getting
refs.
```

3. Once you close the editor, you will see rebase's progress through Git. As predicted, the commit message editor will not open; Git will just rebase the changes into two commits on top of the `orgin/stable-3.1` branch.

```
$ git rebase --interactive
[detached HEAD 70b4eb7] Do not close ArchiveOutputStream on
error
Author: Jonathan Nieder <jrn@google.com>
 6 files changed, 537 insertions(+), 2 deletions(-)
 create mode 100644
org.eclipse.jgit.test/tst/org/eclipse/jgit/api/DescribeComma
```

```
ndTest.java
  create mode 100644
org.eclipse.jgit/src/org/eclipse/jgit/api/DescribeCommand.ja

va
[detached HEAD c5bc5cc] Prepare 3.2.0-SNAPSHOT builds
Author: Matthias Sohn <matthias.sohn@sap.com>
 67 files changed, 422 insertions(+), 372 deletions(-)
 rewrite org.eclipse.jgit.http.server/META-INF/MANIFEST.MF
(61%)
 rewrite org.eclipse.jgit.junit/META-INF/MANIFEST.MF (73%)
 rewrite org.eclipse.jgit.pgm.test/META-INF/MANIFEST.MF
(61%)
 rewrite org.eclipse.jgit.pgm/META-INF/MANIFEST.MF (63%)
 rewrite org.eclipse.jgit.test/META-INF/MANIFEST.MF (77%)
 rewrite org.eclipse.jgit.ui/META-INF/MANIFEST.MF (67%)
 rewrite org.eclipse.jgit/META-INF/MANIFEST.MF (64%)
Successfully rebased and updated refs/heads/rebaseExample3.
```

4. Another difference is that the commit message from the two commits we marked with fixup has disappeared. So, if you compare this with the previous example, it is very clear what the difference is; it is shown in the following command:

```
commit c5bc5cc9e0956575cc3c30c3be4aecab19980e4d
Author: Matthias Sohn <matthias.sohn@sap.com>
Date:   Thu Oct 3 17:40:22 2013 +0200
```

Prepare 3.2.0-SNAPSHOT builds

Change-Id: Iac6cf7a5bb6146ee3fe38abe8020fc3fc4217584
Signed-off-by: Matthias Sohn matthias.sohn@sap.com

5. Finally, we can also confirm that we still have the same source code, but with different commits. This can be done by comparing this commit with the commit we created via 9c96a65, using the following command:

```
$ git diff 9c96a65
```

As predicted, there is no output from `git diff`, so we still have the same

source code.

This check can also be performed on the previous example.

Changing the author of commits using a rebase

When I start working on a new project, I often forget to set my author name and author e-mail address for the specified project; therefore, I often have commits in my local branch that have been committed with the wrong username and/or e-mail ID. Unfortunately, I can't use the same account everywhere as some work with regards to my cooperate account still needs to be done; however, for most other parts, I can use my private account.

Getting ready

Before we begin this exercise, we need a branch, as always with Git. Name the branch `resetAuthorRebase` and make it track `origin/master`. Use the following command to achieve this:

```
$ git checkout -b resetAuthorRebase -t origin/master
Branch resetAuthorRebase set up to track remote branch master
from origin.
Switched to a new branch 'resetAuthorRebase'
```

How to do it...

Now, we want to change the author of all the commits from `origin/stable-3.2` to our `HEAD`, which is `master`. This is just an example; you will rarely have to change the author of commits that have already been published to a remote repository.

You can change the author of the `HEAD` commit by using `git commit --amend --reset-author`; however, this will only change the author of `HEAD` and leave the rest of the commits as they were. We will start by changing the author of the `HEAD` commit and then verifying why that is wrong by performing the following steps:

1. Change the author of the `HEAD` commit as follows:

```
$ git commit --amend --reset-author  
[resetAuthorRebase b0b2836] Update Kepler target platform  
to use Kepler SR2 orbi  
t R-build  
1 file changed, 1 insertion(+), 1 deletion(-)
```

- Verify that you have changed it with the Git log command:

```
$ git log --format='format:%h %an <%ae>' origin/stable-  
3.2..HEAD  
b0b2836 Rasmus Voss <rasmus.voss@live.dk>  
b9a0621 Matthias Sohn <matthias.sohn@sap.com>  
ba15d82 Matthias Sohn matthias.sohn@sap.com
```

- We will list all the commits from `origin/stable-3.2` to `HEAD` and we will define a format with `%h` as the abbreviated commit hash, `%an` for the author's name, and `%ae` for the author's e-mail address. From the output, you can see that I am now the author of the `HEAD` commit, but what we really wanted was to change the author of all the commits. To do this, we will rebase onto the `origin/stable-3.2` branch; then, for each commit, we will stop to amend and reset the author. Git can do most of that work with `--exec` option for the `git rebase`, as follows:

```
git rebase --interactive --exec "git commit --amend --  
reset-author" origin/stable-3.2  
pick b14a939 Prepare 3.3.0-SNAPSHOT builds  
exec git commit --amend --reset-author  
pick f2abbd0 archive: Prepend a specified prefix to all  
entry filenames  
exec git commit --amend --reset-author
```

- As you can see, Git has opened the rebase's to-do list for you, and between every commit, you have the `exec` keyword and the command we specified on the command line. You can have more `exec` lines between commits if you have a use case for them. Closing the editor will start the rebase.
- As you will see, this process is not very good as the commit message editor opens every time and you have to close the editor to allow Git to continue with the rebase. To stop the rebase, clear the commit message editor and Git will return to the command line; then, you can use `git rebase --abort` as follows:

```
Executing: git commit --amend --reset-author
Aborting commit due to empty commit message.
Execution failed: git commit --amend --reset-author
You can fix the problem, and then run
```

```
    git rebase --continue
$ git rebase --abort
```

To achieve what we really want, you can add the `--reuse-message` option for `git commit`; this will reuse the commit message for the commit you will specify. We want to use the message of `HEAD` as we are going to amend it to the `HEAD` commit. So, try again as shown in the following command:

```
git rebase --interactive --exec "git commit --amend --
reset-author --reuse-message=HEAD" origin/stable-3.2
Executing: git commit --amend --reset-author --reuse-
message=HEAD
[detached HEAD 0cd3e87] Prepare 3.3.0-SNAPSHOT builds
 51 files changed, 291 insertions(+), 291 deletions(-)
 rewrite org.eclipse.jgit.junit/META-INF/MANIFEST.MF
(62%)
 rewrite org.eclipse.jgit.junit/META-INF/MANIFEST.MF (73%)
 rewrite org.eclipse.jgit.pgm.test/META-INF/MANIFEST.MF
(61%)
 rewrite org.eclipse.jgit.test/META-INF/MANIFEST.MF (76%)
 rewrite org.eclipse.jgit.ui/META-INF/MANIFEST.MF (67%)
Executing: git commit --amend --reset-author --reuse-
message=HEAD
[detached HEAD faaf25e] archive: Prepend a specified prefix
to all entry filenames
 5 files changed, 115 insertions(+), 1 deletion(-)
Executing: git commit --amend --reset-author --reuse-
message=HEAD
[detached HEAD cfd743e] [CLI] Add option --millis / -m to
debug-show-dir-cache command
Successfully rebased and updated
refs/heads/resetAuthorRebase.
```

6. Git provides an output indicating that the action was a success; however, to verify, you can execute the previous Git log command and you should see the e-mail address has changed on all the commits, as shown in the following command:

```
$ git log --format='format:%h %an <%ae>' origin/stable-  
3.2..HEAD  
9b10ff9 Rasmus Voss <rasmus.voss@live.dk>  
d8f0ada Rasmus Voss <rasmus.voss@live.dk>  
53df2b7 Rasmus Voss rasmus.voss@live.dk
```

How it works...

It works as you would expect! There is one thing to remember: when using the `exec` option, Git will check the work area for unstaged and staged changes. Consider the following command line:

```
exec echo rainy_day > weather_main.txtexec echo sunny_day >  
weather_california.txt
```

If you have a line as illustrated in the preceding command, the first `exec` would be executed and you will then have an unstaged change in your work area. Git would complain and you have to solve that before continuing with the next `exec`. So, if you want to do something like this, you must create a single `exec` line that executes all the things you want. Besides this, the rebase functionality is fairly simple, as it just tries to apply the changes in the order that is specified in the rebase's to-do list. Git will only apply the changes specified in the list, so if you remove some of them, they will not be applied. This is a way to clean up a feature branch for unwanted commits, for instance, commits that enable you to debug.

Auto-squashing commits

When I work with Git, I often create a lot of commits for a single bug fix, but when making the delivery to the remote repository, I prefer and recommend to deliver the bug fix as one commit. This can be achieved with an interactive rebase, but since this should be a common workflow, Git has a built-in feature called autosquash that will help you squash the commits together.

Getting ready

Before we begin with this exercise, we will create a branch from `origin/master` so we are ready to add commits to our fix.

Let's start with something like this:

```
$ git checkout -b readme_update_developer --track origin/master
Branch readme_update_developer set up to track remote branch
master from origin.
Switched to a new branch 'readme_update_developer'
```

How to do it...

After checking the branch, we will create the first commit that we want to squash other commits to. We need to use the abbreviated commit hash from this commit to automatically create other commits that will squash to this commit by performing the following steps:

1. Start by echoing some text into `README.md`:

```
$ echo "More information for developers" >> README.md
```

2. This will append more information to `README.md` for developers; verify that the file has changed using the `git status` as follows:

```
$ git status
On branch readme_update_developer
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: README.md
```

```
no changes added to commit (use "git add" and/or "git
commit -a")
```

- Now, we want to add and commit this; we can do this with the `commit` command with the `-a` flag, which will add any unstaged changes to the commit, as shown in the following command:

```
$ git commit -a -m "Updating information for developers"
[readme_update_developer d539645] Updating information for
developers
 1 file changed, 1 insertion(+)
```

- After you create the commit, remember the abbreviated commit hash; I have highlighted it in bold in the command output. The abbreviation will be different in your environment, and you must have your own abbreviation once you finish the exercise.
- To continue, we will add three commits to the branch, and we would like to squash two of these with the first commit, as shown in the following command:

```
$ echo "even More information for developers" >> README.md
$ git commit -a --squash=d539645 --no-edit
[readme_update_developer d62922d] squash! Updating
information for developers
 1 file changed, 1 insertion(+)
```

- This is the first commit. Notice why we needed to store the abbreviated hash of the first commit; we used it with the `--squash` option for `git commit`. This option will create the commit with the subject of the commit specified. It will also add "squash!" to the start of the subject. This is to indicate that Git should squash this commit when performing a rebase. Now, create the second commit, as shown in the following command:

```
$ echo "even More information for developers" >> README.md
$ git commit -a --squash=d539645 --no-edit
[readme_update_developer 7d6194d] squash! Updating
information for developers
```

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

7. We have added two commits that we would like to squash with the first commit. When committing, I also used the --no-edit option; this will skip the opening of the commit's message editor. If you leave the flag out, the editor will open as it usually does when committing. The difference is that the commit subject has already been set, and you only need to write the commit message. Now, we will create the last commit; we don't want to squash this commit:

```
$ echo "Adding configuration information" >> README.md
$ git commit -a -m "Updating information on configuration"
[readme_update_developer fd07857] Updating information on
configuration
```

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

8. Now, we add the final commit that does not have anything to do with the first three commits we added. This is why we did not use the --squash option. We can now squash the commits together using `git rebase -i`:

```
$ git rebase -i
```

9. You will get the rebase's to-do list up in the configured commit editor. What we would have expected was to have Git configure a squash for the commits we wanted to squash, as shown in the following command:

```
pick d539645 Updating information for developers
pick d62922d squash! Updating information for developers
pick 7d6194d squash! Updating information for developers
pick fd07857 Updating information on configuration
```

10. What you can see is that Git inserted `squash` to the subject of two of the commits, but besides this, we did not get what we had expected. Git requires that you specify `--autosquash` to the `git rebase -i` command. Close the editor and Git will just perform the rebase and will give the following output:

```
Successfully rebased and updated
refs/heads/readme_update_developer.
```

11. Let's try again with `--autosquash` and see what happens with the

rebase's to-do list:

```
$ git rebase -i --autosquash
pick d539645 Updating information for developers
squash d62922d squash! Updating information for developers
squash 7d6194d squash! Updating information for developers
pick fd07857 Updating information on configuration
```

12. Now the rebase's to-do list looks much more like what we expected it to look like. Git has preconfigured the to-do list to show which commits it will squash and which commits it will keep.
13. Closing the to-do list now will start the rebase, and we don't want that. This is because if you clear the to-do list and close the editor, the rebase will be aborted, which is what we want now. The output will be as follows:

Nothing to do

14. What we really want to do is just run `git rebase -i` and Git will use `--autosquash` as a default. This can be achieved with `git config rebase.autosquash true`; try it and then run `git rebase -i`:

```
$ git config rebase.autosquash true
$ git rebase -i
```

15. The rebase's to-do list pops up, and we have the expected result as follows:

```
pick d539645 Updating information for developers
squash d62922d squash! Updating information for developers
squash 7d6194d squash! Updating information for developers
pick fd07857 Updating information on configuration
```

16. Now close the editor and allow the rebase to start. The editor opens and you can change the commit message for the combined message as shown in the following command:

```
# This is a combination of 3 commits.
# The first commit's message is:
Updating information for developers

# This is the 2nd commit message:
```

```
squash! Updating information for developers
```

```
# This is the 3rd commit message:
```

```
squash! Updating information for developers
```

17. Modify the message and close the editor; Git continues with the rebase and completes with the following message:

```
[detached HEAD baadd53] Updating information for
developers
 1 file changed, 3 insertions(+)
Successfully rebased and updated
refs/heads/readme_update_developer.
Verify the commit log with git log -3
$ git log -3
commit 6d83d44645e330d0081d3679aca49cd9bc20c891
Author: Rasmus Voss <rasmus.voss@schneider-electric.com>
Date:   Wed May 21 10:52:03 2014 +0200
```

Updating information on configuration

```
commit baadd53018df2f6f3cdf88d024c3b9db16e526cf
Author: Rasmus Voss <rasmus.voss@schneider-electric.com>
Date:   Wed May 21 10:25:43 2014 +0200
```

```
Updating information for developers
commit 6d724dcd3355f09e3450e417cf173fcfaee9e08
Author: Shawn Pearce <spearce@spearce.org>
Date:   Sat Apr 26 10:40:30 2014 -0700
```

18. As expected, we now have two commits on top of the origin/master commit.

Hopefully, this can assist you when you are just committing away but want to deliver the code as one commit.

There's more...

If you want to avoid opening the commit message editor like in step 17 of the *Auto-squashing commits* recipe, you can use `--fixup= d539645`. This will instead use the commit message from the first commit and totally disregard any message written in the commits.

Chapter 5. Storing Additional Information in Your Repository

In this chapter, we will cover the following topics:

- Adding your first Git note
- Separating notes by category
- Retrieving notes from the remote repository
- Pushing Git notes to a remote repository
- Tagging commits in the repository

Introduction

Git is powerful in many ways. One of the most powerful features of Git is that it has immutable history. This is powerful because nobody can squeeze something into the history of Git without it being noticed by the people who have cloned the repository. This also causes some challenges for developers as some developers would like to change the commit messages after a commit has been released. This is a feature that is possible in many other version control systems but because of the immutable history with Git, it has Git notes, which is essentially an extra `refs/notes/commits` reference in Git. Here, you add additional information to the commits that can be listed when running a `git log` command. You can also release the notes into a remote repository so that people can fetch the notes.

Adding your first Git note

We will add some extra information to the already released code. If we were doing it in the actual commits, we would see the commit hashes change.

Getting ready

Before we start, we need a repository to work in; you can use the previous clone of `jgit`, but to get an output from the example that's almost identical, you can clone the `jgit` repository as follows:

```
$ git clone https://git.eclipse.org/r/jgit/jgit chapter5  
$ cd chapter5
```

How to do it...

We start by creating a local branch `notesMessage` tracking `origin/stable-3.2`. Then, we will try and change the commit message and see that the commit hash changes:

1. Checkout the branch `notesMessage` tracking `origin/stable-3.2`:

```
$ git checkout -b notesMessage --track origin/stable-3.2  
Branch notesMessage set up to track remote branch stable-  
3.2 from origin.  
Switched to a new branch 'notesMessage'
```

2. List the commit hash of `HEAD` of your branch:

```
$ git log -1  
commit f839d383e6fbbda26729db7fd57fc917fa47db44  
Author: Matthias Sohn <matthias.sohn@sap.com>  
Date:   Wed Dec 18 21:16:13 2013 +0100
```

`Prepare post 3.2.0 builds`

```
Change-Id: Ie2bfdee0c492e3d61d92acb04c5bef641f5f132f  
Signed-off-by: Matthias Sohn <matthias.sohn@sap.com>
```

3. Change the commit message by amending the commit using `git commit --amend`. Add a line above the `Change-Id:` line with "Update

MANIFEST files":

```
$ git commit --amend
```

4. Now, we list the commit again and see that the commit hash has changed:

```
$ git log -1
commit 5ccc9c90d29badb1bd860d29860715e0becd3d7b
Author: Matthias Sohn <matthias.sohn@sap.com>
Date:   Wed Dec 18 21:16:13 2013 +0100
```

Prepare post 3.2.0 builds

Update MANIFEST files

Change-Id: Ie2bfdee0c492e3d61d92acb04c5bef641f5f132f
Signed-off-by: Matthias Sohn matthias.sohn@sap.com

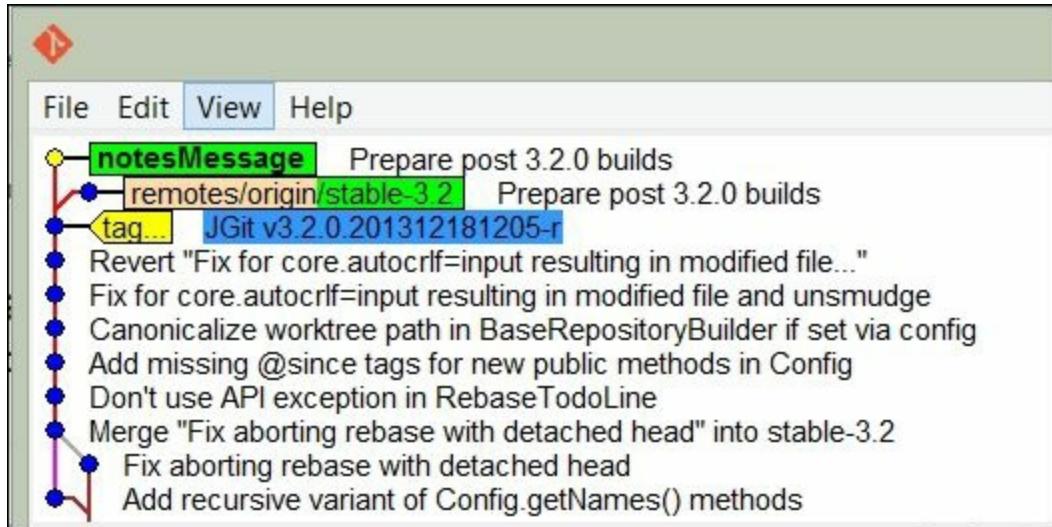
5. Notice that the commit parts have changed from f839d383e6fbnda26729db7fd57fc917fa47db44 to 5ccc9c90d29badb1bd860d29860715e0becd3d7b, as the commit is derived from the content in the commit, the parents of the commit, and the commit message. So, the commit hash will change when updating the commit message. Since we have changed the content of the HEAD commit, we are no longer based on the HEAD commit of the origin/stable-3.2 branch. This becomes visible in gitk and git status:

```
$ git status
On branch notesMessage
Your branch and 'origin/stable-3.2' have diverged,
and have 1 and 1 different commit each, respectively.
(use "git pull" to merge the remote branch into yours)

nothing to commit, working directory clean
```

6. As you can see from the output, our branch has diverged from origin/stable-3.2; this is also visible from Gitk. Notice that I specified which branches and commits I want to see with gitk. In this case, I want to see origin/stable-3.2 and HEAD:

```
$ gitk origin/stable-3.2 HEAD
```



7. To prevent this result, we can add a note to the commit message. Let's start by resetting the branch to `origin/stable-3.2` and then adding a note to the commit:

```
$ git reset --hard origin/stable-3.2
HEAD is now at f839d38 Prepare post 3.2.0 builds
```

8. Now, add the same message as the previous one but just as a note:

```
$ git notes add -m "Update MANIFEST files"
```

9. We have added the note directly from the command line without invoking the editor by using the `-m` flag and then a message. The log will now be visible when running `git log`:

```
$ git log -1
```

```
commit f839d383e6fbdbda26729db7fd57fc917fa47db44
Author: Matthias Sohn <matthias.sohn@sap.com>
Date:   Wed Dec 18 21:16:13 2013 +0100
```

```
Prepare post 3.2.0 builds
```

```
Change-Id: Ie2bfdee0c492e3d61d92acb04c5bef641f5f132f
Signed-off-by: Matthias Sohn <matthias.sohn@sap.com>
```

Notes:

```
Update MANIFEST files
```

10. As you can see from the log output, we have a `Notes:` section with our note. Although it does not add the note directly in the commit

message as the `--amend` option does, we still have our important addition to the commit message. We can verify with `git status` that we have no longer diverged:

```
$ git status
On branch notesMessage
Your branch is up-to-date with 'origin/stable-3.2'.

nothing to commit, working directory clean
```

There's more...

So, you have your notes for your commit and now you want to add to them. You will perhaps expect that you just add the note again with more information. This is not the case. You have the option to append, edit, or force the note to be created:

1. Start by trying to add the note again with additional information:

```
$ git notes add -m "Update MANIFESTS files for next
version"
error: Cannot add notes. Found existing notes for object
f839d383e6fbnda26729db7
fd57fc917fa47db44. Use '-f' to overwrite existing notes
```

2. As predicted, we cannot add the note but we can do it with the `-f` flag:

```
$ git notes add -f -m "Update MANIFESTS files for next
version"
Overwriting existing notes for object
f839d383e6fbnda26729db7fd57fc917fa47db44
```

3. Git overwrites the existing notes due to the `-f` flag. You can also use `--force`, which is the same. Verify it with `git log`:

```
$ git log -1
commit f839d383e6fbnda26729db7fd57fc917fa47db44
Author: Matthias Sohn <matthias.sohn@sap.com>
Date:   Wed Dec 18 21:16:13 2013 +0100
```

Prepare post 3.2.0 builds

Change-Id: Ie2bfdee0c492e3d61d92acb04c5bef641f5f132f

Signed-off-by: Matthias Sohn <matthias.sohn@sap.com>

Notes:

Update MANIFESTS files for next version

4. You can also append a current note with `git notes append`:

```
$ git notes append -m "Verified by Rasmus Voss"
```

5. There is no output from this unless something goes wrong, but you can verify this by using `git log` again. To keep the output to a minimum, we are using `--oneline`. This will show a minimum output of the commit. But to show the note, we have to add `--notes`, which will show the notes for the commits in the output:

```
$ git log -1 --notes --oneline  
f839d38 Prepare post 3.2.0 builds
```

Notes:

Update MANIFESTS files for next version

Verified by Rasmus Voss

6. As we can see from the output, we have the line appended to the note. If you try to use the `edit` option, you will see that you can only use this with the `-m` flag. This makes good sense as you should edit the note and not overwrite or append an already created note:

```
$ git notes edit -m "Rasmus Voss"  
The -m/-F/-c/-C options have been deprecated for the 'edit'  
subcommand.  
Please use 'git notes add -f -m/-F/-c/-C' instead.
```

7. As you can see, Git rejects doing and mentions other ways of doing it.

Tip

The `git notes add` and `git notes edit` commands without any arguments will do exactly the same, that is, open the configured editor and allow you to write a note to the commit.

Separating notes by category

As we saw in the previous example, we can add notes to the commits, but in some cases, it makes sense to store the information sorted by categories, such as `featureImplemented`, `defect`, and `alsoCherryPick`. As briefly explained at the beginning of the chapter, notes are stored in `refs/notes/commits` but we can add multiple references so that we can easily sort and list the different scopes of the notes.

Getting ready

To start this example, we need a new branch that tracks the `origin/stable-3.1` branch; we name the branch `notesReferences`, and create and checkout the branch with the following command:

```
$ git checkout -b notesReferences --track origin/stable-3.1
Branch notesReferences set up to track remote branch stable-3.1
from origin.
Switched to a new branch 'notesReferences'
```

How to do it...

Imagine a situation where we have corrected a defect and did everything we could to ensure the quality of the commit before releasing it. Nonetheless, we had to make another fix for the same defect.

So, we want to add a note to the reference `refs/notes/alsoCherryPick`, which should indicate that if you cherry pick this commit, you should also cherry pick the other commits as they fix the same defect.

In this example, we will find a commit and add some extra information to the commit in multiple notes' reference specifications:

1. Start by listing the top ten commits on the branch so we have something to copy and paste from:

```
$ git log -10 --oneline
da6e87b Prepare post 3.1.0 builds
16ca725 JGit v3.1.0.201310021548-r
```

```
c6aba99 Fix order of commits in rebase todo file header
5a2a222 Prepare post 3.1.0 RC1 builds
6f0681e JGit v3.1.0.201309270735-rc1
a065a06 Attempt to fix graph layout when new heads are
introduced
b4f07df Prepare re-signing pgm's ueberjar to avoid
SecurityException
aa4bbc6 Use full branch name when getting ref in
BranchTrackingStatus
570bba5 Ignore bitmap indexes that do not match the pack
checksum
801aac5 Merge branch 'stable-3.0'
```

2. Now, to add a note for the b4f07df commit in the ref alsoCherryPick, we must use the --ref option for git notes. This has to be specified before the add option:

```
$ git notes --ref alsoCherryPick add -m "570bba5" b4f07df
```

3. No output means success when adding notes. Now that we have a note, we should be able to list it with a single git log -1 command. However, this is not the case. You actually need to specify that you want to list the notes from the specific ref. This can be done with the --notes=alsoCherryPick option for git log:

```
$ git log -1 b4f07df357fccdff891df2a4fa5c5bd9e83b4a4a --
notes=alsoCherryPick
commit b4f07df357fccdff891df2a4fa5c5bd9e83b4a4a
Author: Matthias Sohn <matthias.sohn@sap.com>
Date:   Tue Sep 24 09:11:47 2013 +0200
```

```
Prepare re-signing pgm's ueberjar to avoid
SecurityException
More output...
Change-Id: Ia302e68a4b2a9399cb18025274574e31d3d3e407
Signed-off-by: Matthias Sohn <matthias.sohn@sap.com>
```

```
Notes (alsoCherryPick):
570bba5
```

4. As you can see from the output, Git shows the alsoCherryPick notes. The reason for not showing is that Git defaults to adding notes into refs/notes/commits. It would be nice if you could just show the alsoCherryPick notes' reference by default. This can be done by configuring Git as follows:

```
$ git config notes.displayRef "refs/notes/alsoCherryPick"
```

5. By configuring this option, you are telling Git to always list these notes. But what about the default notes? Have we overwritten the configuration to list the default `refs/notes/commits` notes? We can check this with `git log -1` to see if we still have the test note displayed:

```
$ git log -1
commit da6e87bc373c54c1cda8ed563f41f65df52bacbf
Author: Matthias Sohn <matthias.sohn@sap.com>
Date:   Thu Oct 3 17:22:08 2013 +0200
```

`Prepare post 3.1.0 builds`

`Change-Id: I306a3d40c6ddb88a16d17f09a60e3d19b0716962`
`Signed-off-by: Matthias Sohn <matthias.sohn@sap.com>`

`Notes:`

`test note`

6. No, we did not overwrite the setting to list notes in the default refs. Knowing that we can have as many `notes.displayRef` configurations as we want, we should add all the refs we want to use in our repo. In some situations, it is even better to just add `refs/notes/*`. This will configure Git to show all the notes:

```
$ git config notes.displayRef 'refs/notes/*'
```

7. If we add another note in `refs/notes/defect` now, we should be able to list it without specifying which notes' reference we want to list when using `git log`. We are adding to the commit that already has a note in the `alsoCherryPick` reference:

```
$ git notes --ref defect add -m "Bug:24435"
b4f07df357fccdff891df2a4fa5c5bd9e83b4a4a
```

8. Now, list the commit with `git log`:

```
$ git log -1 b4f07df357fccdff891df2a4fa5c5bd9e83b4a4a
commit b4f07df357fccdff891df2a4fa5c5bd9e83b4a4a
Author: Matthias Sohn <matthias.sohn@sap.com>
Date:   Tue Sep 24 09:11:47 2013 +0200
```

`Prepare re-signing pgm's ueberjar to avoid`

SecurityException

See <http://dev.eclipse.org/mhonarc/lists/jgit-dev/msg02277.html>

Change-Id: Ia302e68a4b2a9399cb18025274574e31d3d3e407

Signed-off-by: Matthias Sohn <matthias.sohn@sap.com>

Notes (alsoCherryPick) :

570bba5

Notes (defect) :

Bug:24435

9. Git shows both notes, which is what we would expect.

How it works...

I have been writing about the `refs/notes/alsoCherryPick` reference and so on. As you know, we refer to the remote branches as references such as `refs/remotes/origin/stable-3.2`, but the local branches are also references such as `refs/heads/develop` for instance.

Since you can create a branch that starts at a specific reference, you should be able to create a branch that starts at the `refs/notes/alsoCherrypick` reference:

1. Create a branch that starts from `refs/notes/alsoCherryPick`. Also, checkout the branch:

```
$ git checkout -b myNotes notes/alsoCherryPick
Switched to a new branch 'myNotes'
```

2. The `myNotes` branch now points to `HEAD` on `refs/notes/alsoCherryPick`. Listing the files on the branch will show a file with the commit hash of the commit we have added the notes to:

```
$ ls
b4f07df357fccdff891df2a4fa5c5bd9e83b4a4a
```

3. Showing the file content will show the text we used as note text:

```
$ cat b4f07df357fccdff891df2a4fa5c5bd9e83b4a4a
570bba5
```

4. As you can see, the abbreviated commit hash 570bba5 we added as a note for b4f07df357fccdff891df2a4fa5c5bd9e83b4a4a is in the file. If we had a longer message, that message would also be shown here.

Retrieving notes from the remote repository

So far, we have been creating notes in our own local repository, which is okay. But if we want to share those notes, we have to be sure to be able to push them. We would also like to be able to retrieve other people's notes from the remote repository. Unfortunately, this is not so plain and simple.

Getting ready

Before we can start, we need another clone from the local clone we already have. This is to show the push and fetch mechanism of Git with git notes:

1. Start by checking out the master branch:

```
$ git checkout master
Checking out files: 100% (1529/1529), done.
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.
```

2. Now, create local branches of all the stable-3.x branches:

```
$ git branch stable-3.0 origin/stable-3.0
Branch stable-3.0 set up to track remote branch stable-3.0
from origin.
$ git branch stable-3.1 origin/stable-3.1
Branch stable-3.1 set up to track remote branch stable-3.1
from origin.
$ git branch stable-3.2 origin/stable-3.2
Branch stable-3.2 set up to track remote branch stable-3.2
from origin.
```

3. We are checking out all these branches because we want to clone this repository and by default all the refs/heads/* branches will be cloned. So, when we clone the chapter5 directory, you will see that we only get the branches we see if you execute git branch:

```
$ git branch
* master
```

```
myNotes  
notesMessage  
notesReference  
stable-3.0  
stable-3.1  
stable-3.2
```

4. Now, go one directory up so you can create your new clone from the chapter5 directory:

```
cd ..  
$ git clone ./chapter5 shareNotes  
Cloning into 'shareNotes'...  
done.
```

5. Now, enter the shareNotes directory and run git branch -a to see that the only remote branches we have are the branches we had checked out as local branches in the chapter5 directory. After this, we are ready to fetch some notes:

```
cd shareNotes  
$ git branch -a  
* master  
  remotes/origin/HEAD -> origin/master  
  remotes/origin/master  
  remotes/origin/myNotes  
  remotes/origin/notesMessage  
  remotes/origin/notesReference  
  remotes/origin/stable-3.0  
  remotes/origin/stable-3.1  
  remotes/origin/stable-3.2
```

6. As predicted, the list matches the Git branch output from the chapter5 directory.

How to do it...

We have now prepared the setup to push and fetch notes. The challenge is that Git is not a default setup to retrieve and push notes, so you won't usually see other people's notes:

1. We start by showing that we did not receive the notes during the clone:

```
$ git log -1 b4f07df357fccdff891df2a4fa5c5bd9e83b4a4a --  
notes=alsoCherryPick  
warning: notes ref refs/notes/alsoCherryPick is invalid  
commit b4f07df357fccdff891df2a4fa5c5bd9e83b4a4a  
Author: Matthias Sohn <matthias.sohn@sap.com>  
Date: Tue Sep 24 09:11:47 2013 +0200
```

Prepare re-signing pgm's ueberjar to avoid
SecurityException

- As expected, the output does not show the note. In the chapter5 directory, we will see the note. To enable the notes to be fetched, we need to create a new fetch rule configuration; it needs to be similar to the fetch rule for refs/heads. Take a look at the configuration from git config:

```
$ git config --get remote.origin.fetch  
+refs/heads/*:refs/remotes/origin/*
```

- This shows that we are fetching refs/heads into the refs/remotes/origin reference, but what we also want to do is fetch refs/notes/* into refs/notes/*:

```
$ git config --add remote.origin.fetch  
'+refs/notes/*:refs/notes/*'
```

- You should now have it configured. If you leave out the --add option from your command, you will overwrite your current settings. Verify that the rule now exists:

```
$ git config --get-all remote.origin.fetch  
+refs/heads/*:refs/remotes/origin/*  
+refs/notes/*:refs/notes/*
```

- Now, try and fetch the notes:

```
$ git fetch  
From c:/Users/Rasmus/repos/.chapter5  
 * branch master      -> FETCH_HEAD  
 * [new ref] refs/notes/alsoCherryPick ->  
refs/notes/alsoCherryPick  
 * [new ref] refs/notes/commits ->  
refs/notes/commits  
 * [new ref] refs/notes/defect -> refs/notes/defect  
 * [new ref] refs/notes/defects ->  
refs/notes/defects
```

- As the Git output indicates, we have received some new refs. So, let us check whether we have the note on the commit now:

```
$ git log -1 b4f07df357fccdff891df2a4fa5c5bd9e83b4a4a --  
notes=alsoCherryPick  
commit b4f07df357fccdff891df2a4fa5c5bd9e83b4a4a  
Author: Matthias Sohn <matthias.sohn@sap.com>  
Date: Tue Sep 24 09:11:47 2013 +0200  
  
        Prepare re-signing pgm's ueberjar to avoid  
SecurityException  
More output...  
Signed-off-by: Matthias Sohn <matthias.sohn@sap.com>  
Notes (alsoCherryPick):  
570bba5
```

- We now have the notes in our repository, which is what we expected.

How it works...

We fetched the notes. The reason why it works is because of the way we fetched. By default, Git is configured to fetch `refs/heads/*` into `refs/remotes/origin/*`. This way, we can easily keep track of what is remote and what is local. The branches in our local repo are in `refs/heads/*`. These branches are also listed when you execute `git branch`.

For notes, we need to fetch `refs/notes/*` into `refs/notes/*` since we want to get the notes from the server and use them with the `git show`, `git log`, and `git notes` Git commands.

Pushing notes to a remote repository

We have tried and succeeded in retrieving the notes from the remote repository, but what about your notes? How can you push them to the server? This has to be done with the push command just as with any other references, such as branches and commits, when you want to publish them to a remote repository.

How to do it...

Before we can push the notes from the `shareNotes` repository, we have to create a note to be pushed, as the notes we have now are all available on the remote repository. The remote repository in this case is the `chapter5` directory:

1. I have found a commit I would like to add a note to, and I want to add the note in the `verified` reference:

```
$ git notes --ref verified add -m "Verified by  
rasmus.voss@live.dk" 871ee53b52a
```

2. Now that we have added the note, we can list it with the `git log` command:

```
$ git log --notes=verified -1 871ee53b52a  
commit 871ee53b52a7e7f6a0fe600a054ec78f8e4bff5a  
Author: Robin Rosenberg <robin.rosenberg@dewire.com>  
Date:   Sun Feb 2 23:26:34 2014 +0100
```

```
    Reset internal state canonical length in  
WorkingTreeIterator when moving
```

```
Bug: 426514  
Change-Id: Ifb75a4fa12291aeeece3dda129a65f0c1fd5e0eb  
Signed-off-by: Matthias Sohn <matthias.sohn@sap.com>
```

```
Notes (verified):
```

```
    Verified by rasmus.voss@live.dk
```

- As expected, we can see the note. If you cannot see the note, you probably missed `--notes=verified` for the `git log` command, since we have not configured `verified` as `notes.displayRef`. To push the note, we must use the `git push` command, because the default push rule in Git is to push branches to `refs/heads/<branchname>`. So, if we just try to push the note to the remote, nothing happens:

```
$ git push  
Everything up-to-date
```

- You will probably see a warning about `git push.default` not being configured; you can safely ignore this for these examples. The important part is that Git shows that everything is up-to-date. But we know we have created a Git note for a commit. So to push these notes, we need to push our note references to the remote notes references. This can be done as follows:

```
$ git push origin refs/notes/*  
Counting objects: 15, done.  
Delta compression using up to 4 threads.  
Compressing objects: 100% (4/4), done.  
Writing objects: 100% (5/5), 583 bytes | 0 bytes/s, done.  
Total 5 (delta 0), reused 0 (delta 0)  
To c:/Users/Rasmus/repos/.chapter5/  
 * [new branch]      refs/notes/verified ->  
   refs/notes/verified
```

- Now something happened; we have a new branch on the remote named `refs/notes/verified`. This is because we have pushed the notes to the remote. What we can do to verify it is go to the `chapter5` directory and check if the `871ee53b52a` commit has a Git note:

```
$ cd ..chapter5/  
$ git log --notes=verified -1 871ee53b52a  
commit 871ee53b52a7e7f6a0fe600a054ec78f8e4bff5a  
Author: Robin Rosenberg <robin.rosenberg@dewire.com>  
Date:   Sun Feb 2 23:26:34 2014 +0100
```

```
    Reset internal state canonical length in  
WorkingTreeIterator when moving
```

Bug: 426514

Change-Id: Ifb75a4fa12291aeeece3dda129a65f0c1fd5e0eb
Signed-off-by: Matthias Sohn <matthias.sohn@sap.com>

Notes (verified) :

Verified by rasmus.voss@live.dk

6. As predicted, we can see the note in this directory.

There's more...

Since Git notes do not work as normal branches, it can be a little difficult to push them back and forth to a repository when you are trying to collaborate on them. Since you cannot just fetch and merge the Git notes branches as easily with other branches, a clear recommendation is to build some tools to add these notes so you only have one server adding the notes.

A very simple but value adding note could be the Jenkins build and test information that published notes about which tests have passed on a commit hash; this is very valid when you have to reopen a defect. You can then actually see in the repository which test has been executed on the commit hash.

Tagging commits in the repository

If you are releasing software with Git, you are bound to deal with tags as the tag describes the different software releases in the repository. There are two types of tags, a lightweight tag and an annotated tag. The lightweight tag is very similar to a branch, since it is just a named reference like `refs/tags/version123`, which points to the commit hash of the commit you are tagging; whereas if it were a branch, it would be `refs/heads/version123`. The difference is the branch moves forward when you work and commit to it. The tag should always point to the same commit hash.

Getting ready

Before we start, you must go to the `chapter5` directory, where we made the original clone for this chapter.

We should start by tagging the commit that is ten commits behind `origin/stable-2.3` and is not a merge. In order to find that commit, we will use the `git log` command.

For the `git log` command, we are using the `--no-merges` option, which will show commits that only have one parent. The `--oneline` option we have used before tells Git to limit the output to one line per commit.

Find the commit as follows:

```
$ git log -11 --no-merges --oneline origin/stable-2.3
49ec6c1 Prepare 2.3.2-SNAPSHOT builds
63dcece JGit v2.3.1.201302201838-r
3b41fcb Accept Change-Id even if footer contains not well-
formed entries
5d7b722 Fix false positives in hashing used by PathFilterGroup
9a5f4b4 Prepare post 2.3.0.201302130906 builds
19d6cad JGit v2.3.0.201302130906
3f8ac55 Replace explicit version by property where possible
1c4ee41 Add better documentation to DirCacheCheckout
e9cf705 Prepare post 2.3rc1 builds
ea060dd JGit v2.3.0.201302060400-rc1
```

60d538f Add getConflictingNames to RefDatabase

How to do it...

Now that we have found the 60d538f commit, we should make it a lightweight tag:

1. Use the `git tag` command to give a meaningful release name:

```
$ git tag 'v2.3.0.201302061315rc1' ea060dd
```

2. Since there is no output, it is a success; to see whether the tag is available, use the `git tag` command:

```
$ git tag -l "v2.3.0.2*"  
v2.3.0.201302061315rc1  
v2.3.0.201302130906
```

3. We are using the `git tag` command with `-l` as a flag, since we want to list the tags and not tag the current `HEAD`. Some repositories have a lot of tags, so to prevent the list from being too long, you can specify which tags you want to list and use a `*` wildcard as I have used. Our tag is available, but all it really says is that we have a tag in the repository with the name `v2.3.0.201302061315rc1`, and if you are using `git show v2.3.0.201302061315rc1`, you will see that the output is the same as `git show ea060dd`:

```
$ git show v2.3.0.201302061315rc1  
commit ea060dd8e74ab588ca55a4fb3ff15dd17343aa88  
Author: Matthias Sohn <matthias.sohn@sap.com>  
Date:   Wed Feb 6 13:15:01 2013 +0100
```

JGit v2.3.0.201302060400-rc1

```
Change-Id: Id1f1d174375f7399cee4c2eb23368d4dbb4c384a  
Signed-off-by: Matthias Sohn <matthias.sohn@sap.com>  
diff --git a/org.eclipse.jgit.ant.test/META-INF/MANIFEST.MF  
b/org.eclipse.jgit.a
```

```
$ git show ea060dd  
commit ea060dd8e74ab588ca55a4fb3ff15dd17343aa88  
Author: Matthias Sohn <matthias.sohn@sap.com>  
Date:   Wed Feb 6 13:15:01 2013 +0100
```

JGit v2.3.0.201302060400-rc1

Change-Id: Id1f1d174375f7399cee4c2eb23368d4dbb4c384a
Signed-off-by: Matthias Sohn <matthias.sohn@sap.com>

diff --git a/org.eclipse.jgit.ant.test/META-INF/MANIFEST.MF
b/org.eclipse.jgit.a

4. There will also be a lot of file diff information in the output but it is exactly the same output. So, in order to add more information, we should use an annotated tag. An annotated tag is a tag where you have to add some information to the tag. To create an annotated tag, we use the --annotate tag for the git tag command:

```
git tag --annotate -m "Release Maturity rate 97%"  
'v2.3.0.201409022257rc2' 1c4ee41
```

5. The -m flag is the same as --message, as I wanted to give the tag a message. If you leave out the -m flag, Git will open the configured editor and you can write a full release note into the annotation of the tag. We can check the tag information with git show:

```
$ git show 'v2.3.0.201409022257rc2'  
tag v2.3.0.201409022257rc2  
Tagger: Rasmus Voss <rasmus.voss@live.dk>  
Date:   Sun Feb 9 22:58:28 2014 +0100
```

Release Maturity rate 97%

```
commit 1c4ee41dc093266c19d4452879afe5c0f7f387f4  
Author: Christian Halstrick christian.halstrick@sap.com
```

6. We can actually see the tag name and information we added with the -m flag. With the lightweight tag, we don't see anything about the tag from the output. We actually don't even see the tag name when using Git show on a lightweight tag.

There's more...

Tags are very powerful as they can add valuable information to the repository, and since tags should be considered official releases in the repository, we should be very careful with them.

Naturally, you can push the tags to a remote area, and contributors to the repository would fetch those tags. This is where you have to be careful. With a legacy version control system, you can go back in time and just change the release, and since these legacy systems are all based on a centralized server where you have to be connected in order to work, changing a release is not that bad, since not so many people use the release or have even downloaded the release. But it is different in Git. If you change a tag that you have already pushed to point to another commit hash, then those developers who have already fetched the tag will not get the new tag unless they delete the tag locally.

1. To prove the dangers of not getting new tag, we will try to delete a tag and recreate it to point to another commit hash:

```
$ git tag -d v1.3.0.201202121842-rc4  
Deleted tag 'v1.3.0.201202121842-rc4' (was d1e8804)
```

2. Now that we have deleted the tag, we are ready to recreate the tag again to point to HEAD:

```
$ git tag -a -m "Local created tag" v1.3.0.201202121842-rc4
```

3. We have recreated the tag and it points to HEAD because we did not specify a commit hash at the end of the command. Now, execute git fetch to see whether you can get the tag overwritten from the remote repository:

```
$ git fetch
```

4. Since there is no output, the tag was probably not overwritten. Let us verify with git show:

```
$ git show v1.3.0.201202121842-rc4  
tag v1.3.0.201202121842-rc4  
Tagger: Rasmus Voss <rasmus.voss@live.dk>  
Date: Sun Feb 9 23:17:18 2014 +0100
```

Local created tag

```
commit 1c4ee41dc093266c19d4452879afe5c0f7f387f4
```

5. As you can see from the output, it is still our locally created tag. To get the tag from the remote again, we need to delete the local tag

and do a Git fetch. To delete a tag, you need to apply the `-d` flag:

```
$ git tag -d v1.3.0.201202121842-rc4
Deleted tag 'v1.3.0.201202121842-rc4' (was 28be24b)
$ git fetch
From https://git.eclipse.org/r/jgit/jgit
 * [new tag]           v1.3.0.201202121842-rc4 ->
   v1.3.0.201202121842-rc4
```

6. As you can see, Git has fetched the tag from the server again. We can verify with `git show`:

```
$ git show v1.3.0.201202121842-rc4
tag v1.3.0.201202121842-rc4
Tagger: Matthias Sohn <matthias.sohn@sap.com>
Date:   Mon Feb 13 00:57:56 2012 +0100

JGit 1.3.0.201202121842-rc4
-----BEGIN PGP SIGNATURE-----
Version: GnuPG/MacGPG2 v2.0.14 (Darwin)

iF4EABEIAAYFAk84UhMACgkQWwXM3hQMKHbwewD/VD62MWCVfLCYUIEZ20C
4Iywx
4O015TedaLFwIOS55HcA/ ipDh6NWFvJdWK3Enm2krjegUNmd9zXT+0pNjt1
J+Pyi
=L Roe
-----END PGP SIGNATURE-----

commit 53917539f822afa12caaa55db8f57c29570532f3
```

7. So, as you can see, we have the correct tag again, but it should also be a warning that once you push a tag to a remote repository, you should never change it, since the developers who are fetching from the repository may never know unless they clone again or delete the tags locally and fetch them again.

In this chapter, we learned how you can tag your commits and add notes to them. This is a powerful method to store additional information after the commit has been committed and published to a shared repository. But before you actually publish your commit, you have the chance to add the most valuable information for a commit. The commit message is where you must specify what you are doing and sometimes why you are doing it.

If you are solving a bug, you should list the bug ID; if you are using a special method to solve the problem, it is recommended that you describe why you have used this awesome technique to solve the problem. So when people look back on your commits, they can also learn a few things on why different decisions were made.

Chapter 6. Extracting Data from the Repository

In this chapter, we will cover the following topics:

- Extracting the top contributor
- Finding bottlenecks in the source tree
- Grepping the commit messages
- The contents of the releases

Introduction

Whether you work in big or small organizations, safeguarding and maintaining data is always important and it keeps track of a fair amount for you, it is just a matter of extracting the data. Some of the data is included in the system by you or any other developer when you fill in the commit message with proper information; for instance, details of the bug you are fixing from the bug tracking system.

The data is not only valid for management, but can also be used to add more time to refactor the .c files, where almost all bugs are fixed.

Extracting the top contributor

Git has a few built-in stats you can get instantaneously. The `git log` command has different options, such as `--numstat`, that will show the number of files added and lines deleted for each file since each commit. However, for finding the top committer in the repository, we can just use the `git shortlog` command.

Getting ready

As with all the examples throughout the book, we are using the `jgit` repository; you can either clone it or go to one of the clones you might already have.

Clone the `jgit` repository as follows:

```
$ git clone https://git.eclipse.org/r/jgit/jgit chapter6  
$ cd chapter6
```

How to do it...

The `shortlog` Git command is very simple and does not leave a lot of options or flags to use with it. It can show the log but in a boiled-down version, and then it can summarize it for us:

1. Start by showing the last five commits with `shortlog`. We can use `-5` to limit the amount of output:

```
$ git shortlog -5  
Jonathan Nieder (1) :  
    Update commons-compress to 1.6  
  
Matthias Sohn (2) :  
    Update com.jcraft.jsch to 0.1.50 in Kepler target  
    platform  
    Update target platforms to use latest orbit build  
  
SATO taichi (1) :  
    Add git checkout --orphan implementation
```

Stefan Lay (1) :

Fix fast forward rebase with rebase.autostash=true

2. As you can see, the output is very different from the `git log` output. You can try it for yourself with `git log -5`. The numbers in parentheses are the number of commits by that committer. Below the name and number are the commit titles of the commits. Note that no commit hashes are shown. To find the top committer with just those five commits is easy, but when you try running `git shortlog` without `-5`, it is hard to find that person. To sort and find the top committer, we can use the `-n` or `--numbered` option to sort the output. The top committer is on top:

```
$ git shortlog -5 --numbered
```

Matthias Sohn (2) :

```
    Update com.jcraft.jsch to 0.1.50 in Kepler target
platform
    Update target platforms to use latest orbit build
```

Jonathan Nieder (1) :

```
    Update commons-compress to 1.6
```

SATO taichi (1) :

```
    Add git checkout --orphan implementation
```

Stefan Lay (1) :

Fix fast forward rebase with rebase.autostash=true

3. As you can see, the output is nicely sorted. If we don't care about the commit subjects, we can use `-s` or `--summary` to only show the commit count for each developer:

```
$ git shortlog -5 --numbered --summary
```

```
 2 Matthias Sohn
  1 Jonathan Nieder
  1 SATO taichi
  1 Stefan Lay
```

4. Finally, we have what we want, except we don't have the e-mail addresses of the committers; this option is also available with `-e` or `--email`. This will also show the e-mail addresses of the committers in the list. This time, we will try it on the entire repository. Currently, we have only listed it for the `HEAD` commit. To list it for the

repository, we need to add `--all` at the end of the command so as to execute the command for all branches:

```
$ git shortlog --numbered --summary --email --all
 765 Shawn O. Pearce <spearce@spearce.org>
 399 Matthias Sohn <matthias.sohn@sap.com>
 360 Robin Rosenberg <robin.rosenberg@dewire.com>
 181 Chris Aniszczyk <caniszczyk@gmail.com>
 172 Shawn Pearce <spearce@spearce.org>
 160 Christian Halstrick <christian.halstrick@sap.com>
 114 Robin Stocker robin@nibor.org
```

5. So, this is the list now; we know who contributed with the most commits, but this picture can be a little skewed as the top committer may just happen to be the creator of the project and may not actively contribute to the repository. So, to list the top committers for the last six months, we can add `--since="6 months ago"` to the `git shortlog` command:

```
$ git shortlog --numbered --summary --email --all --
since="6 months ago"
 73 Matthias Sohn <matthias.sohn@sap.com>
 15 Robin Stocker <robin@nibor.org>
 14 Robin Rosenberg <robin.rosenberg@dewire.com>
 13 Shawn Pearce <sop@google.com>
 12 Stefan Lay <stefan.lay@sap.com>
   8 Christian Halstrick <christian.halstrick@sap.com>
   7 Colby Ranger cranger@google.com
```

6. As you can see, the picture has changed since the start of the repository.

Tip

You can use "n weeks ago", "n days ago", "n months ago", "n hours ago", and so on for specifying time periods. You can also use specific dates such as "1 october 2013".

You can also list the top committer for a specific month using the `--until` option, where you can specify the date you wish to list the commit until. This can be done as follows:

```
$ git shortlog --numbered --summary --email --all --
```

```

since="30 september" --until="1 november 2013"
 15 Matthias Sohn <matthias.sohn@sap.com>
   4 Kaloyan Raev <kaloyan.r@zend.com>
   4 Robin Rosenberg <robin.rosenberg@dewire.com>
   3 Colby Ranger <cranger@google.com>
   2 Robin Stocker <robin@nibor.org>
   1 Christian Halstrick <christian.halstrick@sap.com>
   1 Michael Nelson <michael.nelson@tasktop.com>
   1 Rüdiger Herrmann <ruediger.herrmann@gmx.de>
   1 Tobias Pfeifer <to.pfeifer@web.de>
   1 Tomasz Zarna tomasz.zarna@tasktop.com

```

- As you can see, we get another list, and it seems like Matthias is the main contributor, at least compared to the initial result. These types of data can also be used to visualize the shift of responsibility in a repository by collecting the data for each month since the repository's initialization.

There's more...

While working with code, it is often useful to know who to go to when you need to perform a fix in the software, especially in an area where you are inexperienced. So, it would be nice to figure out who is the code owner of the file or the files you are changing. The obvious reason is to get some input on the code, but also to know who to go to for a code review. You can again use `git shortlog` to figure this out. You can use the command on the files as well:

- To do this, we simply add the file to the end of the `git shortlog` command:

```

$ git shortlog --numbered --summary --email ./pom.xml
 86 Matthias Sohn <matthias.sohn@sap.com>
  21 Shawn O. Pearce <spearce@spearce.org>
   4 Chris Aniszczyk <caniszczyk@gmail.com>
   4 Jonathan Nieder <jrn@google.com>
   3 Igor Fedorenko <igor@ifedorenko.com>
   3 Kevin Sawicki <kevin@github.com>
   2 Colby Ranger cranger@google.com

```

- As for `pom.xml`, we also have a top committer. As all the options you have for `git log` are available for `shortlog`, we can also do this on a directory.

```
$ git shortlog --numbered --summary --email  
./org.eclipse.jgit.console/  
 57 Matthias Sohn <matthias.sohn@sap.com>  
 11 Shawn O. Pearce <spearce@spearce.org>  
   9 Robin Rosenberg <robin.rosenberg@dewire.com>  
   2 Chris Aniszczyk <caniszczyk@gmail.com>  
   1 Robin Stocker robin@nibor.org
```

3. As you can see, it is fairly simple to get some indication on who to go to for the different files or directories in Git.

Finding bottlenecks in the source tree

Often, the development teams know where the bottleneck in the source tree is, but it can be challenging to convince the management that you need resources to rewrite some code. However, with Git, it is fairly simple to extract that type of data from the repository.

Getting ready

Start by checking out the `stable-3.1` release:

```
$ git checkout stable-3.1
Branch stable-3.1 set up to track remote branch stable-3.1 from
origin.
Switched to a new branch 'stable-3.1'
```

How to do it...

We want to start by listing some stats for one commit, and then we can extend the examples to larger chunks of commits:

1. The first option we will be using is `--dirstat` for `git log`:

```
$ git log -1 --dirstat
commit da6e87bc373c54c1cda8ed563f41f65df52bacbf
Author: Matthias Sohn <matthias.sohn@sap.com>
Date:   Thu Oct 3 17:22:08 2013 +0200
```

`Prepare post 3.1.0 builds`

```
Change-Id: I306a3d40c6ddb88a16d17f09a60e3d19b0716962
Signed-off-by: Matthias Sohn <matthias.sohn@sap.com>
```

```
5.0% org.eclipse.jgit.http.server/META-INF/
6.9% org.eclipse.jgit.http.test/META-INF/
3.3% org.eclipse.jgit.java7.test/META-INF/
4.3% org.eclipse.jgit.junit.http/META-INF/
6.6% org.eclipse.jgit.junit/META-INF/
5.5% org.eclipse.jgit.packaging/
5.9% org.eclipse.jgit.pgm.test/META-INF/
```

```
13.7% org.eclipse.jgit.pgm/META-INF/  
15.4% org.eclipse.jgit.test/META-INF/  
 3.7% org.eclipse.jgit.ui/META-INF/  
13.1% org.eclipse.jgit/META-INF/
```

2. The --dirstat option shows which directories have changed in the commit and how much they have changed compared to each other. The default setting is to count the number of lines added to or removed from the commit. So, rearranging the code potentially does not count for any change as the line count might be the same. You can compensate for this slightly by using --dirstat=lines. This option will look at each file line by line and see whether they have changed compared to the previous version:

```
$ git log -1 --dirstat=lines  
commit da6e87bc373c54c1cda8ed563f41f65df52bacbf  
Author: Matthias Sohn <matthias.sohn@sap.com>  
Date: Thu Oct 3 17:22:08 2013 +0200
```

Prepare post 3.1.0 builds

Change-Id: I306a3d40c6ddb88a16d17f09a60e3d19b0716962
Signed-off-by: Matthias Sohn <matthias.sohn@sap.com>

```
4.8% org.eclipse.jgit.http.server/META-INF/  
6.5% org.eclipse.jgit.http.test/META-INF/  
3.2% org.eclipse.jgit.java7.test/META-INF/  
4.0% org.eclipse.jgit.junit.http/META-INF/  
6.1% org.eclipse.jgit.junit/META-INF/  
6.9% org.eclipse.jgit.packaging/  
5.7% org.eclipse.jgit.pgm.test/META-INF/  
13.0% org.eclipse.jgit.pgm/META-INF/  
14.6% org.eclipse.jgit.test/META-INF/  
 3.6% org.eclipse.jgit.ui/META-INF/  
13.8% org.eclipse.jgit/META-INF/
```

3. This also gives a slightly different result. If you would like to limit the output to only show directories with a certain percentage or higher, we can limit the output as follows:

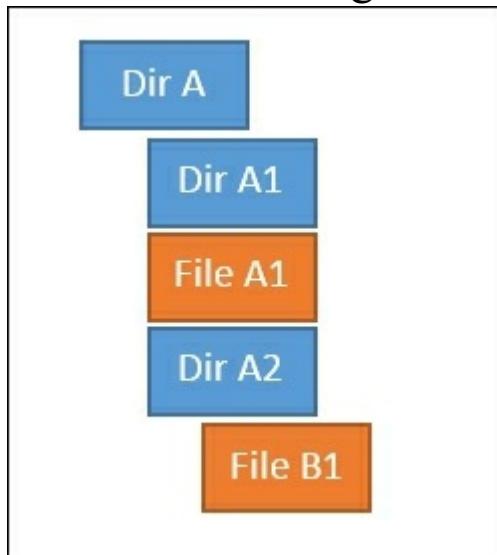
```
$ git log -1 --dirstat=lines,10  
commit da6e87bc373c54c1cda8ed563f41f65df52bacbf  
Author: Matthias Sohn <matthias.sohn@sap.com>  
Date: Thu Oct 3 17:22:08 2013 +0200
```

Prepare post 3.1.0 builds

Change-Id: I306a3d40c6ddb88a16d17f09a60e3d19b0716962
Signed-off-by: Matthias Sohn <matthias.sohn@sap.com>

13.0% org.eclipse.jgit.pgm/META-INF/
14.6% org.eclipse.jgit.test/META-INF/
13.8% org.eclipse.jgit/META-INF/

4. By adding 10 to the --dirstat=lines command, we are asking Git to only show the directories that have 10 percent or higher changes; you can use any number you like here. By default, Git does not count the changes in the subdirectories, but only the files in the directory. So, in this diagram, only changes in File A1 are counted as changes. For the Dir A1 directory and the File B1 file, it is counted as a change in Dir A2:



5. To cumulate this, we can add cumulative to the --dirstat=files,10 command, and this will cumulate the changes and calculate a percentage. Be aware that the percentage can go beyond 100 due to the way it is calculated:

```
$ git log -1 --dirstat=files,10,cumulative
commit da6e87bc373c54c1cda8ed563f41f65df52bacbf
Author: Matthias Sohn <matthias.sohn@sap.com>
Date:   Thu Oct 3 17:22:08 2013 +0200
```

Prepare post 3.1.0 builds

Change-Id: I306a3d40c6ddb88a16d17f09a60e3d19b0716962
Signed-off-by: Matthias Sohn <matthias.sohn@sap.com>

31.3% org.eclipse.jgit.packaging/

6. As you can see, the output is slightly different from what we have seen earlier. By using `git log --dirstat`, you can get some information about what goes on in the repository. Obviously, you can also do this for all the commits between two releases or two commit hashes. Let's try this, but instead of using `git log`, we will be using `git diff`, as Git will show the accumulated diff between the two releases, and `git log` will show the dirstat for each commit between the releases:

```
$ git diff origin/stable-3.1..origin/stable-3.2 --dirstat
 4.0% org.eclipse.jgit.packaging/org.eclipse.jgit.target/
 3.9% org.eclipse.jgit.pgm.test/tst/org/eclipse/jgit/pgm/
 4.1% org.eclipse.jgit.pgm/
20.7% org.eclipse.jgit.test/tst/org/eclipse/jgit/api/
21.3%
org.eclipse.jgit.test/tst/org/eclipse/jgit/internal/storage
/file/
 5.2% org.eclipse.jgit.test/tst/org/eclipse/jgit/
14.5% org.eclipse.jgit/src/org/eclipse/jgit/api/
 6.5% org.eclipse.jgit/src/org/eclipse/jgit/lib/
 3.9% org.eclipse.jgit/src/org/eclipse/jgit/transport/
 4.6% org.eclipse.jgit/src/org/eclipse/jgit/
```

7. So, between the `origin/stable-3.1` and `origin/stable-3.2` branches, we can see which directories have the highest percentage of changes. We can then dig a little deeper using `--stat` or `--numstat` for the directory, and again use `git diff`. We will also use `--relative="org.eclipse.jgit.test/tst/org/eclipse/"`, which will show the relative path of the files from `org.eclipse.jgit.test/tst/org/eclipse/`. This will look better in the console. Feel free to try this without using the following option:

```
$ git diff --pretty origin/stable-3.1..origin/stable-3.2 -
-numstat --relative
="org.eclipse.jgit.test/tst/org/eclipse/jgit/internal/"
org.eclipse.jgit.test/
tst/org/eclipse/jgit/internal/
4          2          storage/file/FileRepositoryBuilderTest.java
```

```

8      1      storage/file/FileSnapshotTest.java
0     741      storage/file/GCTest.java
162     0      storage/file/GcBasicPackingTest.java
119     0      storage/file/GcBranchPrunedTest.java
119     0      storage/file/GcConcurrentTest.java
85      0      storage/file/GcDirCacheSavesObjectsTest.java
104     0      storage/file/GcKeepFilesTest.java
180     0      storage/file/GcPackRefsTest.java
120     0      storage/file/GcPruneNonReferencedTest.java
146     0      storage/file/GcReflogTest.java
78      0      storage/file/GcTagTest.java
113     0      storage/file/GcTestCase.java

```

8. The first number is the number of lines added, and the second number is the lines removed from the files between the two branches.

There's more...

We have used `git log`, `git diff`, and `git shortlog` to find information about the repository, but there are so many options for those commands on how to find bottlenecks in the source code.

If we want to find the files with the most commits, and these are not necessarily the files with the most line additions or deletions, we can use `git log`:

1. We can use `git log` between the `origin/stable-3.1` and `origin/stable-3.2` branches and list all the files changed in each commit. Then, we just need to sort and accumulate the result with some bash tools:

```

$ git log origin/stable-3.1..origin/stable-3.2 --
format=format: --name-only

org.eclipse.jgit.ant.test/META-INF/MANIFEST.MF
org.eclipse.jgit.ant.test/pom.xml

```

2. First, we are just executing the command without the use of the bash tools. You can see from the extensive output that you only see file names and nothing else. This is due to the options used. The `--format=format: option` tells Git to not display any commit-message-

related information, and --name-only tells Git to list the files for each commit. Now all we have to do is count them:

```
$ git log origin/stable-3.1..origin/stable-3.2 --  
format=format: --name-only | sed '/^$/d' | sort | uniq -c  
| sort -r | head -10  
12 se.jgit/src/org/eclipse/jgit/api/RebaseCommand.java  
12 est/tst/org/eclipse/jgit/api/RebaseCommandTest.java  
9 org.eclipse.jgit/META-INF/MANIFEST.MF  
7 org.eclipse.jgit.pgm.test/META-INF/MANIFEST.MF  
7 org.eclipse.jgit.packaging/pom.xml  
6 pom.xml  
6 pse.jgit/src/org/eclipse/jgit/api/RebaseResult.java  
6 org.eclipse.jgit.test/META-INF/MANIFEST.MF  
6 org/eclipse/jgit/pgm/internal/CLIText.properties  
6 org.eclipse.jgit.pgm/META-INF/MANIFEST.MF
```

3. Now we have a list of the top ten files between the two releases, but before we proceed further, let's just go through what we did. We got the list of files, and we used `sed '/^$/d'` to remove empty lines from the output. After this, we used `sort` to sort the list of files. Then, we used `uniq -c`, which counts the occurrences of each item in the files and adds the number from the output. Finally, we sorted in reverse order using `sort -r` and displayed only the top ten results using `head 10`. To proceed from here, we should list all the commits between the branches that are changing the top file:

```
$ git log origin/stable-3.1..origin/stable-3.2  
org.eclipse.jgit/src/org/eclipse  
/jgit/api/RebaseCommand.java  
commit e90438c0e867bd105334b75df3a6d640ef8dab01  
Author: Stefan Lay <stefan.lay@sap.com>  
Date:   Tue Dec 10 15:54:48 2013 +0100  
  
Fix aborting rebase with detached head  
  
Bug: 423670  
Change-Id: Ia6052867f85d4974c4f60ee5a6c820501e8d2427  
  
commit f86a488e32906593903acb31a93a82bed8d87915
```

4. By adding the file to the end of the `git log` command, we will see the commits between the two branches. Now all we have to do is to grep commits that have the bug, so we can tell our manager the

number of bugs we fixed in this file.

Grepping the commit messages

Now we know how to list and sort files that we make frequent changes to and vice versa, but we are also interested in finding out the bugs we are fixing, the features we are implementing, and perhaps who is signing the code. All this information is usually available in the commit message. Some companies have a policy that you need to have a referral to a bug, a feature, or some other reference in the commit message. By having this information in the commit message, it is a lot easier to produce a nice release note as well.

Getting ready

As we will mostly be grepping the Git database in these examples, we really don't need to check something out or be at a specific commit for this example. So, if you are still lurking around in the `chapter6` folder, we can continue.

How to do it...

Let's see how many commits in the repository are referring to a bug:

1. First of all, we need to know the pattern for bugs referred to in the commit messages. I did this by looking in the commits, and the pattern for Jgit is to use `Bug:` 6 digits; so, to find all of these commits, we use the `--grep` option for `git log`, and we can grep for "`[Bb] [Uu] [gG] : [0-9] \+ :`"

```
$ git log --all --grep="^ [bB] [uU] [gG] : [0-9] "
```

```
commit 3db6e05e52b24e16fbe93376d3fd8935e5f4fc9b
```

```
Author: Stefan Lay <stefan.lay@sap.com>
```

```
Date: Wed Jan 15 13:23:49 2014 +0100
```

```
Fix fast forward rebase with rebase.autostash=true
```

```
The folder .git/rebase-merge was not removed in this  
case. The
```

```
repository was then still in rebase state, but neither  
abort nor
```

continue worked.

Bug: 425742

Change-ID: I43cea6c9e5f3cef9d6b15643722fddecb40632d9

2. You should get a lot of commits as output, but you should notice all the commits have a referral to a bug ID. So what was the grep doing? The `^ [Bb] [Uu] [gG] :` part matches any combination of lowercase and uppercase bugs. The `^` character means from the beginning of the line. The `:` character is matching `:`. Then, we have `[0-9] \+,` which will match any number between zero and nine, and the `\+` part means one or more occurrences. But enough with regular expressions for grep. We have a lot of output (which is valuable), but for now, we just want to count the commits. We can do this by piping it to `wc -l` (`wordcount -l` is to count the lines):

```
$ git log --all --oneline --grep="^ [bB] [uU] [gG] : [0-9] \+" |  
wc -l  
366
```

3. Before piping it to `wc`, remember to use `--oneline` to limit the output to one line for each commit. As you can see, when I was writing this, Jgit has reference to 366 bugs that have all been fixed and released into the repository. If you are used to using regular expressions in another scripting or programming language, you will see that using `--grep` does not support everything. You can enable a more extensive regular expression support using the `--extended-regexp` option for `git log`; however, the pattern still has to be used with `--grep`:

```
$ git log --all --oneline --extended-regexp --grep="^ [bB]  
[Uu] [gG] : [0-9] {6}"  
3db6e05 Fix fast forward rebase with rebase.autostash=true  
c6194c7 Update com.jcraft.jsch to 0.1.50 in Kepler target  
platform  
1def0a1 Fix for core.autocrlf=input resulting in modified  
file and unsmudge  
0ce61ca Canonicalize worktree path in BaseRepositoryBuilder  
if set via config  
e90438c Fix aborting rebase with detached head  
2e0d178 Add recursive variant of Config.getNames() methods
```

4. I have used it in the preceding example, and you can see we are

getting the same commits. I have used a slightly different expression, and have now added `{6}` instead of `\+ the {6}`, which searches for six occurrences of the associated pattern; in our case, it is six digits as it is next to the `[0-9]` pattern. We can verify by counting the lines or commits again with `wc -l`:

```
$ git log --all --oneline --extended-regexp --grep="^ [bB][uU] [gG]: [0-9]{6}" | wc -l  
366
```

5. We get the same number. To shrink the regular expression even more, we can use `--regexp-ignore-case`, which will ignore the case for the pattern:

```
$ git log --all --oneline --regexp-ignore-case --extended-regexp --grep="^bug: [0-9]{6}"  
3db6e05 Fix fast forward rebase with rebase.autostash=true  
c6194c7 Update com.jcraft.jsch to 0.1.50 in Kepler target  
platform  
1def0a1 Fix for core.autocrlf=input resulting in modified  
file and unsmudge  
0ce61ca Canonicalize worktree path in BaseRepositoryBuilder  
if set via config  
e90438c Fix aborting rebase with detached head  
2e0d178 Add recursive variant of Config.getNames() methods
```

6. Now we have the exact same output, and we no longer have `[bB]` `[uU]` `[Gg]` but just `bug`.

Now you know how to grep the commit messages for information, and you can grep for anything in the commit message and list all the commits where the regular expression matches.

The contents of the releases

While extracting information from Git, one of the natural things to do is to generate release notes. To generate a release note, you need all the valid information from the repository between this release and the previous release.

We can utilize some of the methods we have used earlier to generate the data we want.

How to do it...

We start by listing the commits between two tags,

v2.3.1.201302201838-r and v3.0.0.201305080800-m7, and then we build on that information:

1. By using `git log` with v3.0.0.201305080800-m7..v3.0.0.201305080800-m7, we will get the commits between the tags:

```
$ git log --oneline v2.3.1.201302201838-
r..v3.0.0.201305080800-m7
00108d0 JGit v3.0.0.201305080800-m7
e27993f Add missing @since tags
d7cc6eb Move org.eclipse.jgit.pgm's resource bundle to
internal package
75e1bdb Merge "URIish: Allow multiple slashes in paths"
b032623 Remove unused repository field from RevWalk
a626f9f Merge "Require a DiffConfig when creating a
FollowFilter"
```

2. As we have a lot of commits between these two tags, let's count them using `wc -l`:

```
$ git log --oneline v2.3.1.201302201838-
r..v3.0.0.201305080800-m7 | wc -l
211
```

3. There are 211 commits between the tags. Now, we will show the most modified files between the releases:

```
$ git log v2.3.1.201302201838-r..v3.0.0.201305080800-m7 --
format=format: --name-only | sed '/^$/d' | sort | uniq -c
```

```

| sort -r | head -10
  11 org.eclipse.jgit/src/org/eclipse/jgit/internal/st
  10 org.eclipse.jgit/src/org/eclipse/jgit/internal/sto
  10 org.eclipse.jgit.pgm/resources/org/eclipse/jgit/p
   9 org.eclipse.jgit.test/META-INF/MANIFEST.MF
   8 pom.xml
   8 org.eclipse.jgit/src/org/eclipse/jgit/storage/pac
   8 org.eclipse.jgit/src/org/eclipse/jgit/internal/sto
   8 org.eclipse.jgit.pgm/src/org/eclipse/jgit/pgm/CLI
   7 org.eclipse.jgit/src/org/eclipse/jgit/storage/dfs/D
   7 org.eclipse.jgit/src/org/eclipse/jgit/storage/dfs/D

```

4. This information is valid as we now have an overview of where the majority of the changes are. Then, we can find the commit that refers to bugs so we can list the bug IDs:

```

$ git log --format=format:%h --regexp-ignore-case --
extended-regexp --grep="bug: [0-9]{6}"
v2.3.1.201302201838-r..v3.0.0.201305080800-m7 | xargs -n1
git log -1 | grep --ignore-case -E "commit [0-9a-f]{40}|bug:"
commit e8f720335f86198d4dc99af10ffb6f52e40ba06f
  Bug: 406722
commit f448d62d29acc996a97ffbbdec955d14fde5c254
  Bug: 388095
commit 68b378a4b5e08b80c35e6ad91df25b1034c379a3
  Bug: 388095
commit 8bd1e86bb74da17f18272a7f2e8b6857c800a2cc
  Bug: 405558
commit 37f0e324b5e82f55371ef8adc195d35f7a196c58
  Bug: 406722
commit 1080cc5a0d67012c0ef08d9468fbbc9d90b0c238
  Bug: 403697
commit 7a42b7fb95ecd2c132b2588e5ede0f1251772b30
  Bug: 403282
commit 78fca8a099bd2efc88eb44a0b491dd8aecc222b0
  Bug: 405672
commit 4c638be79fde7c34ca0fcaad13d7c4f1d9c5ddd2
  Bug: 405672

```

5. So, what we have here is a nice list of the bugs being fixed and their corresponding commit hashes.

How it works...

We are using some bash tools to get this list of fixed bugs. I will briefly explain what they are doing in this section:

- The `xargs -n1 git log -1` part will execute `git log -1` on each commit coming from the first `git log` command, `git log --format=format:%h --regexp-ignore-case --extended-regexp --grep="bug: [0-9]{6}" v2.3.1.201302201838-r..v3.0.0.201305080800-m7.`
- The `grep --ignore-case -E "commit [0-9a-f]{40} | bug:"` grep will ignore the case in the regular expression and `-E` will enable an extended regular expression. You might see that a lot of these options for the tool grep are the same options we have for `git log`. The regular expression is matching commit and 40 characters with the `[0-9a-f]` range or `bug:`. The `|` character means or. Remember we are in the output from `git log -1`.

All of this information we have extracted is the basis for a good, solid release note, with information on what has changed from one release to another.

The next natural step would be to look into the bug tracking system and also list the titles for each error being fixed in the commits. However, that is not something we will go through here as it all depends on the system you are using.

Chapter 7. Enhancing Your Daily Work with Git Hooks, Aliases, and Scripts

In this chapter, we will cover the following topics:

- Using a branch description in the commit message
- Creating a dynamic commit message
- Using external information in the commit message
- Preventing the push of specific commits
- Configuring and using Git aliases
- Configuring and using Git scripts
- Setting up and using a commit template

Introduction

In order to function in a corporate environment, there should be certain prerequisites to the code that is produced. It should be able to compile and pass specific sets of unit tests. There should also be certain documentation in the commit messages, such as references to a bug fix ID or an instance. Most of these are just scripts that are executed, so why not put these items in to the process? In this chapter, you will see some examples of how to transfer data from one location to the commit message before you see the message. You will also learn how you can verify whether you are pushing your code to the right location. Finally, you will see how you can add scripts to Git.

A hook in Git is a script that will be executed on different events, such as pushing, committing, or rebasing. These scripts, if they exit with a non-zero value, cancel the current Git operation. You can find these hook scripts in the `.git/hooks` folder in any Git clone. If they have the `.sample` file extension, then they are not active.

Using a branch description in the commit message

In [Chapter 3, Branching, Merging, and Options](#), we mentioned that you can set a description to your branch, and this information can be retrieved from a script using the `git config --get branch.<branchname> description` command. In this example, we will take this information and use it for the commit message.

We will be using the `prepare-commit-msg` hook. The `prepare-commit-msg` hook is executed every time you want to commit, and the hook can be set to anything you wish to check for before you actually see the commit message editor.

Getting ready

We need a clone and a branch to get started on this exercise, so we will clone `jgit` again to the `chapter7.5` folder:

```
$ git clone https://git.eclipse.org/r/jgit/jgit chapter7.5
Cloning into 'chapter7.5'...
remote: Counting objects: 2170, done
remote: Finding sources: 100% (364/364)
remote: Total 45977 (delta 87), reused 45906 (delta 87)
Receiving objects: 100% (45977/45977), 10.60 MiB | 1.74 MiB/s,
done.
Resolving deltas: 100% (24651/24651), done.
Checking connectivity... done.
Checking out files: 100% (1577/1577), done.
```

Checkout a local `descriptioInCommit` branch that tracks the `origin/stable-3.2` branch:

```
$ cd chapter7.5
$ git checkout -b descriptioInCommit --track origin/stable-3.2
Branch descriptioInCommit set up to track remote branch stable-3.2 from origin.
Switched to a new branch 'descriptioInCommit'
```

How to do it...

We will start by setting the description for our local branch. Then, we will create the hook that can extract this information and put it in the commit message.

We have our local `descriptioInCommit` branch for which we need to set a description. We will use the `--edit-description` Git branch to add a description to our local branch. This opens the description editor, and you can type in a message by performing the following steps:

1. When you execute the command, the description editor will open and you can type in a message:

```
$ git branch --edit-description descriptioInCommit
```

2. Now, type in the following message:

```
Remote agent not connection to server
```

```
When the remote agent is trying to connect  
it will fail as network services are not up  
and running when remote agent tries the first time
```

3. You should write your branch description just as you write your commit messages. It makes more sense only then to reuse the description in the commit. Now, we will verify whether we have a message with the following description:

```
$ git config --get branch.descriptioInCommit.description  
Remote agent not connection to server
```

```
When the remote agent is trying to connect  
it will fail as network services are not up  
and running when remote agent tries the first time
```

4. As expected, we have the desired output. Now, we can continue with creating the hook that will take the description and use it.

Next, we will check if we have a description for the hook, and if we do, we will use that description as the commit message.

5. First, we will ensure that we can get the information into the commit

message at our desired position. There are many ways to do this, and I have settled on the following one: open the prepare-commit-msg hook file and type in the following script:

```
#!/bin/bash
BRANCH=$(git branch | grep '\*' | sed 's/\*\//g' | sed 's/
//g')
DESCRIPTION=$(git config --get
branch.${BRANCH}.description)
echo $BRANCH
#echo "$DESCRIPTION"
if [ -z "$DESCRIPTION" ]; then
    echo "No desc for branch using default template"
else
    # using tr to convert newlines to #
    # else sed will have a problem.
    DESCRIPTION=$(echo "$DESCRIPTION" | tr -s '\n' '#')
    # replacing # with \n
    DESCRIPTION=$(echo "$DESCRIPTION" | sed 's/#/\n/g')
    # replacing the first \n with \n\n
    DESCRIPTION=$(echo "$DESCRIPTION" | sed 's/\n/\n\n/')
    echo "$DESCRIPTION"
    using
    sed -i "1 i$DESCRIPTION" $1
fi
```

6. Now, we can try to create a commit and see whether the message is being displayed as predicted. Use `git commit --allow-empty` to generate an empty commit but also to trigger the prepare-commit-msg hook:

```
$ git commit --allow-empty
```

7. You should get the message editor with our branch description as the commit message as follows:

Remote agent not connection to server

**When the remote agent is trying to connect
it will fail as network services are not up
and running when remote agent tries the first time**

```
# Please enter the commit message for your changes. Lines
starting
```

```
# with '#' will be ignored, and an empty message aborts the
commit.
# On branch descriptionInCommit
# Your branch is up-to-date with 'origin/stable-3.2'.
#
# Untracked files:
#       hen the remote agent is trying to connect
#
```

8. This is as we expected. Save the commit message and close the editor. Try using the `git log -1` command to verify whether we have the following message in our commit:

```
$ git log -1
commit 92447c6aac2f6d675f8aa4cb88e5abdfa46c90b0
Author: Rasmus Voss <rasmus.voss@live.dk>
Date:   Sat Mar 15 00:19:35 2014 +0100
```

Remote agent not connection to server

When the remote agent is trying to connect
it will fail as network services are not up
and running when remote agent tries the first time

9. You should get something similar to a commit message that is the same as our branch description. However, what about an empty branch description? How will our hook handle that? We can try again with a new branch named `noDescriptionBranch`, use `git checkout` to create it, and check it, as shown in the following command:

```
$ git checkout -b noDescriptionBranch
Switched to a new branch 'noDescriptionBranch'
```

10. Now, we will make yet another empty commit to see whether the commit message will be as follows:

```
$ git commit --allow-empty
```

11. You should get the commit message editor with the default commit message text as follows:

```
# Please enter the commit message for your changes. Lines
starting
# with '#' will be ignored, and an empty message aborts the
```

```
commit.  
# On branch noDescriptionBranch  
# Untracked files:  
#       hen the remote agent is trying to connect  
#
```

This is all as we expected. This script can be combined with the next exercise that will take content from a defect system as well.

Creating a dynamic commit message template

Developers can be encouraged to do the right thing, or developers can be forced to do the right thing; however, in the end, developers need to spend time coding. So, if a good commit message is required, we can use the prepare-commit-msg hook to assist the developer.

In this example, we will create a commit message for developers that contains information about the state of the work area. It will also insert some information from a web page; this could just as well be defect information from Bugzilla for instance.

Getting ready

To start with this exercise, we will not be cloning a repository, but we will be creating one. For doing this, we will be using `git init`, as shown in the following code. You can use `git init <directory>` to create a new repository somewhere, or you can also go to a directory and execute `git init`, and Git will create a repository for you.

```
$ git init chapter7
Initialized empty Git repository in
c:/Users/Rasmus/repos/chapter7/.git/
$ cd chapter7
```

How to do it...

We have our `chapter7` directory where we just initialized our repository. In this directory, the hooks are already available. Just look into the `.git/hooks` directory. We will be using the `prepare-commit-msg` hook. Perform the following steps:

1. Start by looking into the folder with the following hooks:

```
$ ls .git/hooks/
applypatch-msg.sample  pre-applypatch.sample
pre-rebase.sample      commit-msg.sample
```

`pre-commit.sample`
`post-update.sample`
`update.sample`

`prepare-commit-msg.sample`
`pre-push.sample`

2. As you can see, there are plenty of hooks in each of the hook files. There is an example script and a small explanation to what the hook does and when it is executed. To enable `prepare-commit-msg`, rename the file as shown in the following code:

```
$ cd .git/hooks/  
$ mv prepare-commit-msg.sample prepare-commit-msg  
$ cd ../../..
```

3. Open the `prepare-commit-msg` file in your preferred editor; I prefer gVim.
4. You can read the information in the file, but for our examples, we will clear the file so that we can include the script.
5. Now include the following command in the file:

```
#!/bin/bash  
echo "I refuse to commit"  
exit 1
```

6. Save the file.
7. Finally, try to commit something or nothing. Usually, you cannot make a commit that is empty, but with the `--allow-empty` option, you can create an empty commit as follows:

```
$ git commit --allow-empty  
I refuse to commit
```

8. As you can see, we get the message we put in the `prepare-commit-msg` script file. You can check whether we don't have a commit by using `git log -1` as follows:

```
$ git log -1  
fatal: bad default revision 'HEAD'
```

There is no commit, and we get an error message that we have not seen before. The message has to be there because there is no commit so far in this repository. Before we make further changes to the script, we should know that the prepare commit message hook takes some arguments depending on the situation. The first

argument is always .git/COMMIT_EDITMSG, and the second argument can be merge, commit, squash, or template, depending on the situation. We can use these in the script.

9. Change the script so that we can reject amending commits as follows:

```
#!/bin/bash
if [ "$2" == "commit" ]; then
    echo "Not allowed to amend"
    exit 1
fi
```

10. Now that we have changed the script, let's create a commit and try to amend it as follows:

```
$ echo "alot of fish" > fishtank.txt
$ git add fishtank.txt
$ git commit -m "All my fishes are belong to us"
[master (root-commit) f605886] All my fishes are belong to us
  1 file changed, 1 insertion(+)
   create mode 100644 fishtank.txt
```

11. Now that we have a commit, let's try to amend it using git commit --amend:

```
$ git commit --amend
Not allowed to amend
```

12. As we expected, we were not allowed to amend the commit. If we wish to extract some information, for instance, from a bug handling system, we will have to put this information in to the file before opening the editor. So, again, we will change the script as follows:

```
#!/bin/bash
if [ "$2" == "commit" ]; then
    echo "Not allowed to amend"
    exit 1
fi
MESSAGE=$(curl http://whatthecommit.com | grep "<p>" | sed 's/<p>//')
sed -i -r "s/# Please/$MESSAGE\n&/" $1:
```

13. This script downloads a commit message from <http://www.whatthecommit.com/> and inserts it into the commit

message; so, every time you commit, you will get a new message from the web page. Let's give it a try by using the following command:

```
$ echo "gravel, plants, and food" >>fishtank.txt  
$ git add fishtank.txt  
$ git commit
```

14. When the commit message editor opens, you should see a message from whatthecommit.com. Close the editor, and using `git log -1`, verify whether we have the commit as follows:

```
git log -1  
commit c087f75665bf516af2fe30ef7d8ed1b775bcb97d  
Author: Rasmus Voss <rasmus.voss@live.dk>  
Date:   Wed Mar 5 21:12:13 2014 +0100
```

640K ought to be enough for anybody

15. As expected, we have succeeded with the commit. Obviously, this is not the best message to have for the committer. However, what I have done for my current employer is listed the bugs assigned to the developer as follows in the commit message:

```
# You have the following artifacts assigned  
# Remove the # to add the artifact ID to the commit message  
  
#[artf23456] Error 02 when using update handler on wlan  
#[artf43567] Enable Unicode characters for usernames  
#[artf23451] Use stars instead of & when keying pword
```

16. This way, the developer can easily select the correct bug ID, or the artefact ID from team Forge in this case, using the correct format for the other systems that will look into the commit messages.

There's more...

You can extend the functionalities of the prepare commit message hook very easily, but you should bear in mind that the waiting time for fetching some information should be worth the benefits. One thing that is usually easy to check is a dirty work area.

Here, we need to use the `git status` command in the prepare commit message hook, and we need to predict whether we will have modified files after the commit:

1. To check this, we need to have something staged for committing and some unstaged changes as follows:

```
$ git status  
On branch master  
nothing to commit, working directory clean
```

2. Now, modify the `fishtank.txt` file:

```
$ echo "saltwater" >> fishtank.txt
```

3. Use `git status --porcelain` to check the work area:

```
$ git status --porcelain  
M fishtank.txt
```

4. Add the file to the staging area using `git add`:

```
$ git add fishtank.txt
```

5. Now try `git status --porcelain`:

```
$ git status --porcelain  
M fishtank.txt
```

6. What you should note is the space before `M` the first time we use the `--porcelain` option for `git status`. The `porcelain` option provides a machine-friendly output that shows the state of the files for `git status`. The first character is the status in the staging area, whereas the second character is the status in the work area. So, `MM` `fishtank.txt` would mean the file is modified in the work area and in the staging area. So, if you modify `fishtank.txt` again, the following is the result you can expect:

```
$ echo "sharks and oysters" >> fishtank.txt  
$ git status --porcelain  
MM fishtank.txt
```

7. As expected, the output from `git status` is `MM` `fishtank.txt`. We can use this in the hook to tell whether the work area will have uncommitted changes after we commit. Add the following command

to the prepare-commit-msg file:

```
for file in $(git status --porcelain)
do
    if [ ${file:1:1} ]; then
        DIRTY=1
    fi
done
if [ "${DIRTY}" ]; then
    sed -i -r "s/# Please/You have a dirty workarea are you
sure you wish to commit \?\n&/" $1
fi
```

8. First, we list all the files that have changed with `git status --porcelain`. Then, for each of these, we check whether there is a second character. If this is true, we will have a dirty work area after the commit. In the end, we just insert the message in the commit message so that it is available for the developer to see. Let's try and commit the change by using the following command:

```
$ git commit
```

9. Note that you have a message like the following one. The first line might be different as we still have the message from <http://www.whatthecommit.com/>:

```
somebody keeps erasing my changes.
You have a dirty workarea are you sure you wish to commit ?
# Please enter the commit message for your changes. Lines
starting
# with '#' will be ignored, and an empty message aborts the
commit.
# On branch master
# Changes to be committed:
#       modified: fishtank.txt
#
# Changes not staged for commit:
#       modified: fishtank.txt
#
```

10. Saving the file and closing the editor will create the commit. Verify with `git log -1` as follows:

```
$ git log -1
commit 70cad5f7a2c3f6a8a4781da9c7bb21b87886b462
```

Author: Rasmus Voss <rasmus.voss@schneider-electric.com>

Date: Thu Mar 6 08:25:21 2014 +0100

somebody keeps erasing my changes.

You have a dirty workarea are you sure you wish to commit ?

11. We have the information we expected. The text about the dirty work area is in the commit message . To clean up nicely before the next exercise, we should reset our work area to HEAD as follows:

```
$ git reset --hard HEAD
```

```
HEAD is now at 70cad5f somebody keeps erasing my changes.
```

Now, it is just a matter of finding out what suits you. Is there any information you would like to check before you commit and potentially push the code to a remote? I can think of the following options:

- Style checks in code
- Pylint to check your Python scripts
- Check for files that you are not allowed to add to Git

There are several other items, probably one for every organization or development team in the world. However, this clearly is one way of taking tedious manual work away from the developer so that he or she can focus on coding.

Using external information in the commit message

The commit hook is executed when you close the commit message editor. It can, among other things, be used to manipulate the commit message or review the commit message by machine to check whether it has a specific format.

In this recipe, we will be manipulating and checking the content of a commit message.

Getting ready

To start this exercise, we just need to create a branch and check it out. We need to disable the current prepare-commit-msg hook; we can do this by simply renaming it. Now, we can start working on the commit-msg hook by using the following command:

```
git checkout -b commit-msg-example
Switched to a new branch 'commit-msg-example'
$ mv .git/hooks/prepare-commit-msg .git/hooks/prepare-commit-
msg.example
```

How to do it...

What we want to do in the first example is to check whether the defect information is correct. There is no need to release a commit that refers to a defect that does not exist:

1. We will start by testing the commit-msg hook. First, make a copy of the current hook, then we will force the hook to exit with a non-zero value that will abort the creation of the commit:

```
$ cp .git/hooks/commit-msg.sample .git/hooks/commit-msg
```

2. Now, open the file in your preferred editor and add the following lines to the file:

```
#!/bin/bash
echo "you are not allowed to commit"
exit 1
```

- Now, we will try to make a commit and see what happens as follows:

```
$ echo "Frogs, scallops, and coco shell" >> fishtank.txt
$ git add fishtank.txt
$ git commit
```

- The editor will open and you can write a small commit message. Then, close the editor. You should see the you are not allowed to commit message, and if you check with git log -1, you will see that you don't have a commit with the message you just used, as follows:

```
you are not allowed to commit
$ git log -1
commit 70cad5f7a2c3f6a8a4781da9c7bb21b87886b462
Author: Rasmus Voss <rasmus.voss@schneider-electric.com>
Date:   Thu Mar 6 08:25:21 2014 +0100
```

```
    somebody keeps erasing my changes.
```

```
    You have a dirty workarea are you sure you wish to
commit ?
```

- As you can see, the commit message hook is executed after you close the message editor, whereas the prepare-commit-msg hook is executed before the message editor. To validate, if we have a proper reference to the hook in our commit message, we will be checking whether a specific error is available for the Jenkins-CI project. Replace the lines in the commit-msg hook so that it looks like the following command:

```
#!/bin/bash
JIRA_ID=$(cat $1 | grep jenkins | sed 's/jenkins //g')
ISSUE_INFO=$(curl https://issues.jenkins-
ci.org/browse/JENKINS-$JIRA_ID | grep "The issue you are
trying to view does not exist.")
if [ "$ISSUE_INFO" ]; then
    echo "Jenkins issue ${JIRA_ID} does not exist"
    echo "Please try again"
    exit 1
else
    TITLE=$(curl https://issues.jenkins-
```

```
ci.org/browse/JENKINS-$JIRA_ID} | grep -E "<title>.*</title>")
    echo "Jenkins issue ${JIRA_ID}"
    echo "${TITLE}"
    exit 0
fi
```

6. We are using Curl to retrieve the web page, and we are using grep to find a line with The issue you are trying to view does not exist; if we find this text, we know that the ID does not exist. Now we should create a commit and see what happens if we put in the wrong ID, jenkins 384895, or an ID that exists as jenkins 3157. To check this, we will create a commit as follows:

```
$ echo "more water" >> fishtank.txt
$ git add fishtank.txt
$ git commit
```

7. In the commit message, write something such as Feature cascading... as a commit message subject. Then, in the body of the commit message, insert jenkins 384895. This is the important part as the hook will use that number to look it up on the Jenkins issue tracker:

Feature: Cascading...

jenkins 384895

8. You should end up with the following output:

```
Jenkins issue 384895 does not exist
Please try again
```

9. This is what we expected. Now, verify with `git status` whether the change has not been committed:

```
$ git status
On branch commit-msg-example
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   fishtank.txt
```

10. Now we will try to commit again; this time, we will be using the correct JIRA ID:

```
$ git commit
```

11. Key in a commit message like the previous one; this time make sure the Jenkins issue ID is one that exists. You can use 3157:

```
Feature: Cascading...
```

```
jenkins 3157
```

12. Saving the commit message should result in an output as follows. We can clean it some more by removing the title HTML tags:

```
<title>[#JENKINS-3157] Feature request: cascading project  
settings - Jenkins JIRA</title>  
[commit-msg-example 3d39ca3] Feature: Cascading...  
1 file changed, 2 insertions(+)
```

13. As you can see, we can get information to output. We could also add this information to the commit message itself. Then, we can change and insert this as the `else` clause in the script:

```
TITLE=$(curl https://issues.jenkins-ci.org/browse/JENKINS-  
${JIRA_ID} | grep -E "<title>.*</title>")  
TITLE=$(echo ${TITLE} | sed 's/^<title>///' | sed  
's/</\title>$/''')  
echo "${TITLE}" >> $1  
echo "Jenkins issue ${JIRA_ID}"  
echo "${TITLE}"  
exit 0
```

14. To test, we will create a commit again, and in the message, we need to specify the JIRA ID that exists:

```
$ echo "Shrimps and mosquitos" >> fishtank.txt  
$ git add fishtank.txt  
$ git commit  
After saving the commit message editor you will get an  
output similar like this.  
Jenkins issue 3157  
[#JENKINS-3157] Feat  
[commit-msg-example 6fa2cb4] More cascading  
1 file changed, 1 insertion(+)
```

15. To verify whether we got the information in the message, we will use `git log -1` again:

```
$ git log -1
```

```
commit 6fa2cb47989e12b05cd2689aa92244cb244426fc
Author: Rasmus Voss <rasmus.voss@schneider-electric.com>
Date: Thu Mar 6 09:46:18 2014 +0100
```

More cascading

```
jenkins 3157
[#JENKINS-3157] Feature request: cascading project
settings - Jenkins JIRA
```

As expected, we have the information at the end of the commit. In these examples, we are just discarding the commit message if the JIRA ID does not exist; this is a little harsh on the developer. So, you can combine this with the prepare-commit-msg hook. So, if commit-msg halts the commit process, then save the message temporarily so that the prepare-commit-msg hook can use that message when the developer tries again.

Preventing the push of specific commits

The pre-push hooks are triggered whenever you use the push command and the script execution happens before the push; so, we can prevent a push if we find a reason to reject the push. One reason could be you have a commit that has the `nopush` text in the commit message.

Getting ready

To use the Git pre-push, we need to have a remote repository for which we will be cloning `jgit` again as follows:

```
$ git clone https://git.eclipse.org/r/jgit/jgit chapter7.1
Cloning into 'chapter7.1'...
remote: Counting objects: 2429, done
remote: Finding sources: 100% (534/534)
remote: Total 45639 (delta 145), reused 45578 (delta 145)
Receiving objects: 100% (45639/45639), 10.44 MiB | 2.07 MiB/s,
done.
Resolving deltas: 100% (24528/24528), done.
Checking connectivity... done.
Checking out files: 100% (1576/1576), done.
```

How to do it...

We want to be able to push to a remote, but unfortunately, Git will try to authenticate through HTTPS for the `jgit` repository before the hooks are executed. Because of this, we will create a local clone from the `chapter7.1` directory. This will make our remote a local folder.

```
$ git clone --branch master ./chapter7.1/ chapter7.2
Cloning into 'chapter7.2'...
done.
Checking out files: 100% (1576/1576), done.
$ cd chapter7.2
```

We are cloning the `chapter7.1` directory in a folder named `chapter7.2`, and checking the master branch when the clone has finished. Now, we

can go back to the chapter7.1 directory and continue with the exercise.

What we now want to do is to create a commit with a commit message that has `nopush` as part of it. By adding this word to the commit message, the code in the hook will automatically stop the push. We will be doing this on top of a branch. So, to start with, you should check out a `prepushHook` branch that tracks the `origin/master` branch and then creates a commit. We will try to push it for the remote when we have the pre-push commit in place, as follows:

1. Start by creating a new branch named `prepushHook` that tracks `origin/master`:

```
$ git checkout -b prepushHook --track origin/master
Branch prepushHook set up to track remote branch master
from origin.
Switched to a new branch 'prepushHook'
```

2. Now, we reset back in time; it is not so important how far back we do this. So, I have just selected a random commit as follows:

```
$ git reset --hard 2e0d178
HEAD is now at 2e0d178 Add recursive variant of
Config.getNames() methods
```

3. Now we can create a commit. We will do a simple inline replace with `sed` and then add `pom.xml` and commit it:

```
$ sed -i -r 's/2.9.1/3.0.0/g' pom.xml
$ git add pom.xml
$ git commit -m "Please nopush"
[prepushHook 69d571e] Please nopush
1 file changed, 1 insertion(+), 1 deletion(-)
```

4. To verify whether we have the commit with the text, run `git log -1`:

```
$ git log -1
commit 1269d14fe0c32971ea33c95126a69ba6c0d52bbf
Author: Rasmus Voss <rasmus.voss@live.dk>
Date:   Thu Mar 6 23:07:54 2014 +0100
```

Please nopush

5. We have what we want in the commit message. Now, we just need

to prep the hook. We will start by copying the sample hook to the real name so that it will be executed on push:

```
cp .git/hooks/pre-push.sample .git/hooks/pre-push
```

6. Edit the hook so that it has the code as shown in the following snippet:

```
#!/bin/bash
echo "You are not allowed to push"
exit 1
```

7. Now, we are ready to push. We will be pushing our current branch HEAD to the master branch in the remote:

```
$ git push origin HEAD:refs/heads/master
You are not allowed to push
error: failed to push some refs to '.../chapter7.1/'
```

8. As expected, the hook is being executed, and the push is being denied by the hook. Now we can implement the check we want. We want to exit if we have the word nopush in any commit message. We can use git log --grep to search for commits with the keyword nopush in the commit message, as shown in the following command:

```
$ git log --grep "nopush"
commit 51201284a618c2def690c9358a07c1c27bba22d5
Author: Rasmus Voss <rasmus.voss@live.dk>
Date:   Thu Mar 6 23:07:54 2014 +0100
```

Please nopush

9. We have our newly created commit with the keyword nopush. Now, we will make a simple check for this in the hook and edit the pre-push hook so that it has the following text:

```
#!/bin/bash
COMMIT=$ (git log --grep "nopush")
if [ "$COMMIT" ]; then
    echo "You have commit(s) with nopush message"
    echo "aborting push"
    exit 1
fi
```

10. Now we can try to push again to see what the result will be. We will

try to push our HEAD to the master branch on the remote origin:

```
$ git push origin HEAD:refs/heads/master
You have commit(s) with nopush message
aborting push
error: failed to push some refs to
'c:/Users/Rasmus/repos/.chapter7.1/'
```

As expected, we are not allowed to push as we have the `nopush` message in the commit.

There's more...

Having a hook that can prevent you from pushing commits that you don't want to push is very handy. You can specify any keywords you want. Words such as `reword`, `temp`, `nopush`, `temporary`, or `hack` can all be things you want to stop, but sometimes you want to get them through anyway.

What you can do is have a small checker that will check for specific words and then list the commits and ask if you want to push anyway.

If you change the script to the following snippet, the hook will try to find commits with the keyword `nopush` and list them. If you wish to push them anyway, then you can find an answer to the question and Git will push anyway.

```
#!/bin/bash
COMMITS=$(git log --grep "nopush" --format=format:%H)
if [ $COMMITS ]; then
    exitmaybe=1
fi
if [ $exitmaybe -eq 1 ]; then
while true
do
    'clear'
    for commit in "$COMMITS"
    do
        echo "$commit has no push in the message"
    done
    echo "Are you sure you want to push the commit(s) "
```

```

read REPLY <&1
case $REPLY in
  [Yy]* ) break;;
  [Nn]* ) exit 1;;
  * ) echo "Please answer yes or no.";;
esac
done
fi

```

Try it with the `git push` command again as shown in the following snippet:

```

$ git push origin HEAD:refs/heads/master
Commit 70fea355bac0c65fd51f4874d75e65b4a29ad763 has nopush in
message
Are you sure you want to push the commit(s)

```

Type `n` and press enter. Then, expect the push to be aborted with the following message:

```

error: failed to push some refs to
'c:/Users/Rasmus/repos/.chapter7.1/'

```

As predicted, it will not push. However, on the other hand, if you press `y`, Git will push to the remote. Try it now using the following command:

```

$ git push origin HEAD:refs/heads/master
054c5f78fdc82141e9d73e6b6955c38ff79c8b2e has no push in the
message
Are you sure you want to push the commit(s)
y
To c:/Users/Rasmus/repos/.chapter7.1/
 ! [rejected]          HEAD -> master (non-fast-forward)
error: failed to push some refs to
'c:/Users/Rasmus/repos/.chapter7.1/'
hint: Updates were rejected because a pushed branch tip is
behind its remote
hint: counterpart. Check out this branch and 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 predicted, the push will be tried, but as you can see from the output,

we are rejected by the remote. This is because we diverged, and the push was not working at the tip of the master branch.

So, with this hook, you can make your life a little easier by having the hook prevent you from accidentally pushing something you are not interested in getting pushed. This example also considers commits that have been released; so, if you select a different keyword, then other commits, not only the locally created ones, will be taken into consideration by the script.

Configuring and using Git aliases

Git aliases, like Unix aliases, are short commands that can be configured on a global level or for each repository. It is a simple way of renaming some Git commands to short abbreviations, for example, `git checkout` could be `git co` and so on.

How to do it...

It is very simple and straightforward to create an alias. You simply need to configure it with `git config`.

What we will do is check a branch and then create its aliases one by one and execute them to view their output by performing the following steps:

1. So, we will start by checking a branch named `gitAlias` that tracks the `origin/stable-3.2` branch:

```
$ git checkout -b gitAlias --track origin/stable-3.2
Branch gitAlias set up to track remote branch stable-3.2
from origin.
Switched to a new branch 'gitAlias'
```

2. After this, we can start creating some aliases. We will start with the following one that will simply just amend your commit:

```
$ git config alias.amm 'commit --amend'
```

3. Executing this alias will open the commit message editor with the following message from the `HEAD` commit:

```
$ git amm
Prepare post 3.2.0 builds
```

```
Change-Id: Ie2bfdee0c492e3d61d92acb04c5bef641f5f132f
Signed-off-by: Matthias Sohn matthias.sohn@sap.com
```

4. As you can see, it can be very simple to speed up the process of your daily workflow with Git aliases. The following command will just work on the last 10 commits using `--oneline` as an option for

```
git log:
```

```
$ git config alias.lline 'log --oneline -10'
```

- Using the alias will give you the following output:

```
$ git lline
314a19a Prepare post 3.2.0 builds
699900c JGit v3.2.0.201312181205-r
0ff691c Revert "Fix for core.autocrlf=input resulting in mo
1def0a1 Fix for core.autocrlf=input resulting in modified f
0ce61ca Canonicalize worktree path in BaseRepositoryBuilder
be7942f Add missing @since tags for new public methods in g
ea04d23 Don't use API exception in RebaseTodoLine
3a063a0 Merge "Fix aborting rebase with detached head" into
e90438c Fix aborting rebase with detached head
2e0d178 Add recursive variant of Config.getNames() methods
```

- You can also perform a simple checkout. Thus, instead of using the Git checkout, you can just as well use `git co <branch>`. Configure it as follows:

```
$ git config alias.co checkout
```

- You will see that the aliases take arguments just as the regular Git command does. Let's try the alias using the following command:

```
$ git co master
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.
$ git co gitAlias
Switched to branch 'gitAlias'
Your branch and 'origin/stable-3.2' have diverged,
and have 1 and 1 different commit each, respectively.
(use "git pull" to merge the remote branch into yours)
```

- The command works as expected. You may wonder why we diverged after checking out the `gitAlias` branch again. Then, we diverged when we amended the `HEAD` commit. This is because the next alias is creating a commit with everything that has not been committed in the work area, except for the untracked files:

```
$ git config alias.ca 'commit -a -m "Quick commit"'
```

- Before we can test the alias, we should create a file and modify it to show what it actually does. So, create a file as shown in the

following command:

```
$ echo "Sharks" > aquarium
$ echo "New HEADERTEXT" >pom.xml
```

10. To verify what you want, run `git status`:

```
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:   pom.xml

Untracked files:
  (use "git add <file>..." to include in what will be
committed)

  aquarium

no changes added to commit (use "git add" and/or "git
commit -a")
```

11. Now we can test the alias using the following command:

```
$ git ca
[gitAlias ef9739d] Quick commit
  1 file changed, 1 insertion(+), 606 deletions(-)
  rewrite pom.xml (100%)
```

12. To verify whether the `aquarium` file was not a part of the commit, use `git status`:

```
Untracked files:
  (use "git add <file>..." to include in what will be
committed)

  aquarium

nothing added to commit but untracked files present (use
"git add" to track)
```

13. You can also use `git log -1 --name-status` to see the commit we just created:

```
$ git log -1
```

```
commit ef9739d0bffe354c75b82f3b785780f5e3832776
Author: Rasmus Voss <rasmus.voss@live.dk>
Date:   Thu Mar 13 00:01:49 2014 +0100
```

Quick commit

14. The output is just as we expected. The next alias is a little different as it will count the number of commits in the repository, and this can be done with the `wc` (wordcount) tool. However, since this is not a built-in Git tool, we have to use the exclamation mark and also specify Git:

```
$ git config alias.count '!git log --all --oneline | wc -l'
```

15. Let's try it with the following command:

```
$ git count
3008
```

16. So, currently, I have 3008 commits in the repository. This also means you can execute external tools as if they were Git tools just by creating a Git alias; for instance, if you are using Windows, you can create an alias as follows:

```
$ git config alias.wa '!explorer .'
```

17. This alias will open up an explorer window at the path you are currently at. The next one shows what changed in the `HEAD` commit. It executes this with the `--name-status` option for `git log`:

```
$ git config alias.g11 'log -1 --name-status'
```

18. Now try it using the following command:

```
$ git g11
commit ef9739d0bffe354c75b82f3b785780f5e3832776
Author: Rasmus Voss <rasmus.voss@live.dk>
Date:   Thu Mar 13 00:01:49 2014 +0100
```

Quick commit

M **pom.xml**

19. As you can see, it simply lists the commit and the files including what happened to the files in the commit. As the aliases take

arguments, we can actually reuse this functionality to list the information for another branch. Let's try it with the following command:

```
$ git gl1 origin/stable-2.1
commit 54c4eb69acf700fdf80304e9d0827d3ea13cbc6d
Author: Matthias Sohn <matthias.sohn@sap.com>
Date:   Wed Sep 19 09:00:33 2012 +0200
```

Prepare for 2.1 maintenance changes

Change-Id: I436f36a7c6dc86916eb4cde038b27f9fb183465a
Signed-off-by: Matthias Sohn <matthias.sohn@sap.com>

```
M      org.eclipse.jgit.ant.test/META-INF/MANIFEST.MF
M      org.eclipse.jgit.ant.test/pom.xml
M      org.eclipse.jgit.ant/META-INF/MANIFEST.MF
M      org.eclipse.jgit.ant/pom.xml
M      org.eclipse.jgit.console/META-INF/MANIFEST.MF
M      org.eclipse.jgit.console/pom.xml
M      org.eclipse.jgit.http.server/META-INF/MANIFEST.MF
M      org.eclipse.jgit.http.server/pom.xml
```

As you can see, we get the expected output. So, for instance, if you have been using a specific set of options for `git diff`, then you can just as well make it an alias to use it with ease.

How it works...

It is as simple as inserting text in the `config` file. So, you can try and open the `.git/config` configuration file, or you can list the configuration with `git config -list`:

```
$ git config --list | grep alias
alias.amm=commit --amend
alias.lline=log --oneline -10
alias.co=checkout
alias.ca=commit -a -m "Quick commit"
alias.count=!git log --all --oneline | wc -l
```

This alias feature is very strong, and the idea behind it is that you should use it to shorten those long one-liners that you use otherwise. You can

also use this feature to cut down those one-liners to small aliases so that you can use the command frequently and with more precision. If you have it as an alias, you will run it the same way every time, where keying a long command is bound to fail once in a while.

Configuring and using Git scripts

Yes, we have aliases, and aliases do what they do best—take small one-liners and convert them into small useful Git commands. However, when it comes to larger scripts that are also a part of your process, and you would like to incorporate them into Git, you can simply name the script `git-scriptname`, and then use it as `git scriptname`.

How to do it...

There are a few things to remember. The script has to be in your path so that Git can use the script. Besides this, only imagination sets the boundary:

1. Open your favorite editor and insert the following lines in the file:

```
#!/bin/bash
NUMBEROFCOMMITS=$(git log --all --oneline | wc -l)
while :
    WHICHCOMMIT=$((RANDOM % ${NUMBEROFCOMMITS} + 1))
    COMMITSUBJECT=$(git log --oneline --all -$WHICHCOMMIT | tail -n1)
    COMMITSUBJECT_=$(echo $COMMITSUBJECT | cut -b0-60)
do
    if [ $RANDOM -lt 14000 ]; then
        printf "\e[1m%-60s \e[32m%-10s\e[m\n"
    "${COMMITSUBJECT_}" ' PASSED'
    elif [ $RANDOM -gt 15000 ]; then
        printf "\e[1m%-60s \e[31m%-10s\e[m\n"
    "${COMMITSUBJECT_}" ' FAILED'
    fi
Done
```

2. Save the file with the name `git-likeaboss`. This is a very simple script that will list random commit subjects with either passed or failed as the result. It will not stop until you press *Ctrl + c*.

```
$ git likeaboss
5ec4977 Create a MergeResult for deleted/modified      PASSED
fcc3349 Add reflog message to TagCommand             PASSED
591998c Do not allow non-ff-rebase if there are ed    PASSED
0d7dd66 Make sure not to overwrite untracked notfil  PASSED
```

```
5218f7b Propagate IOException where possible where      FAILED
f5fe2dc Teach PackWriter how to reuse an existing s      FAILED
```

3. Note you can also tab complete these commands, and Git will take them into consideration when you slightly misspell commands as follows:

```
$ git likeboss
git: 'likeboss' is not a git command. See 'git --help'.

Did you mean this?
  likeaboss
```

I know this script in itself is not so useful in a day-to-day environment, but I hope you get the point I am trying to make. All scripts revolve around the software delivery chain and you can just as well name them Git as they are part of Git. This makes it much easier to remember which scripts you have available for your job.

Note

Both Git aliases and Git scripts will show up as Git commands while using tab completion. Type in `git <tab> <tab>` to see the list of possible Git commands.

Setting up and using a commit template

In this chapter, we have been using dynamic templates, but Git also has the option of a static commit template, which essentially is just a text file configured as a template. Using the template is very easy and straightforward.

Getting ready

First of all, we need a template. This has to be a text file whose location you should know. Create a file with the following content:

```
#subject no more than 74 characters please  
  
#BugFix id in the following formats  
#artf [123456]  
#PCP [AN12354365478]  
#Bug: 123456  
#Descriptive text about what you have done  
#Also why you have chosen to do in that way as  
#this will make it easier for reviewers and other  
#developers.
```

This is my take on a simple commit message template. You might find that there are other templates out there that prefer to have the bug in the title or at the bottom of the commit message. The reason for having this at the top is that people often tend not to read the important parts of the text! The important part here is the formatting of the references to systems outside Git. If we get these references correct, we can automatically update the defect system as well. Save the files as ~/committemplate.

How to do it...

We will configure our newly created template, and then we will make a commit that will utilize the template.

To configure the template, we need to use `git config commit.template <pathToFile>` to set it, and as soon as it is set, we can try to create a commit and see how it works:

1. Start by configuring the template as follows:

```
$ git config commit.template ~/committemplate
```

2. Now list the `config` file to see that it has been set:

```
$ git config --list | grep template  
commit.template=c:/Users/Rasmus/committemplate
```

3. As we predicted, the configuration was a success. The template, just as any other configuration, can be set at a global level using `git config --global`, or it can be set at a local repository level by leaving out the `--global` option. We configured our commit template for this repository only. Let's try and make a commit:

```
$ git commit --allow-empty
```

4. Now the commit message editor should open, and you should see our template in the commit message editor:

```
#subject no more than 74 characters please
```

```
#BugFix id in the following formats  
#artf [123456]  
#PCP [AN12354365478]  
#Bug: 123456  
#Descriptive text about what you have done  
#Also why you have chosen to do in that way as  
#this will make it easier for reviewers and other  
#developers.
```

It is really as simple as that.

So, in this chapter, we have seen how we can prevent pushing when we have special words in our commit messages. We have also seen how you can dynamically create a commit message with valid information for you or another developer when you are committing. We have also seen how we can build functionality into your own Git by adding small scripts or aliases that all are executed using the Git way. I hope this information

will help you to work smarter instead of harder.

Chapter 8. Recovering from Mistakes

In this chapter, we will cover the following topics:

- Undo – remove a commit completely
- Undo – remove a commit and retain the changes to files
- Undo – remove a commit and retain the changes in the staging area
- Undo – working with a dirty area
- Redo – recreate the latest commit with new changes
- Revert – undo the changes introduced by a commit
- Reverting a merge
- Viewing past Git actions with `git reflog`
- Finding lost changes with `git fsck`

Introduction

It is possible to correct mistakes made in Git the with `git push` context (without exposing them if the mistake is found before sharing or publishing the change). If the mistake is already pushed, it is still possible to undo the changes made to the commit that introduced the mistake.

We'll also look at the `reflog` command and how we can use that and `git fsck` to recover lost information.

There is no `git undo` command in core Git. One of the reasons being ambiguity on what needs to be undone, for example, the last commit, the added file, and so on. If you want to undo the last commit, how should that be done? Should the changes introduced to the files by the commit be deleted? For instance, just roll back to the last known good commit, or should they be kept so that could be changed for a better commit or should the commit message simply be reworded?. In this chapter, we'll explore the possibilities to undo a commit in several ways depending on what we want to achieve. We'll explore four ways to undo a commit:

- Undo everything; just remove the last commit like it never happened
- Undo the commit and unstage the files; this takes us back to where we were before we started to add the files
- Undo the commit, but keep the files in the index or staging area so we can just perform some minor modifications and then complete the commit
- Undo the commit with the dirty work area

Note

The undo and redo commands in this chapter are performed on commits that are already published in the example repository. You should usually *not* perform the undo and redo commands on commits that are already published in a public repository, as you will be rewriting history. However, in the following recipes, we'll use an example repository and execute the operations on published commits so that everyone can have the same experience.

Undo – remove a commit completely

In this example, we'll learn how we can undo a commit as if it had never happened. We'll learn how we can use the `reset` command to effectively discard the commit and thereby reset our branch to the desired state.

Getting ready

In this example, we'll use the example of the `hello world` repository, clone the repository, and change your working directory to the cloned one:

```
$ git clone https://github.com/dvaske/hello_world_cookbook.git  
$ cd hello_world_cookbook
```

How to do it...

First, we'll try to undo the latest commit in the repository as though it never happened:

1. We'll make sure our working directory is clean, no files are in the modified state, and nothing is added to the index:

```
$ git status  
On branch master  
Your branch is up-to-date with 'origin/master'.  
  
nothing to commit, working directory clean
```

2. Also, check what is in our working tree:

```
$ ls  
HelloWorld.java Makefile          hello_world.c
```

3. If all works well, we'll check the log to see the history of the repository. We'll use the `--oneline` switch to limit the output:

```
$ git log --oneline  
3061dc6 Adds Java version of 'hello world'  
9c7532f Fixes compiler warnings
```

5b5d692 Initial commit, K&R hello world

4. The most recent commit is the 3061dc6 Adds Java version of 'hello world' commit. We will now undo the commit as though it never happened and the history won't show it:

```
$ git reset --hard HEAD^  
HEAD is now at 9c7532f Fixes compiler warnings
```

5. Check the log, status, and filesystem so that you can see what actually happened:

```
$ git log --oneline  
9c7532f Fixes compiler warnings  
5b5d692 Initial commit, K&R hello world  
$ 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  
$ ls  
hello_world.c
```

6. The commit is now gone along with all the changes it introduced (Makefile and HelloWorld.java).

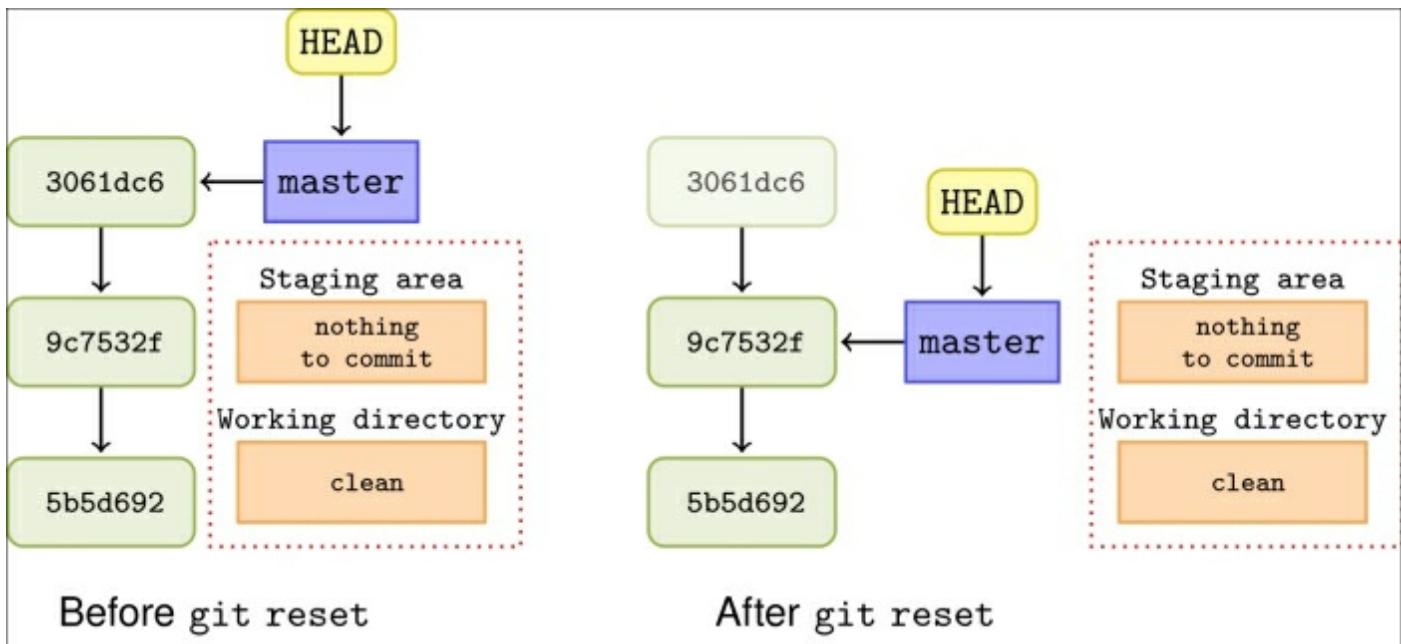
Note

In the last output of `git status`, you can see that our `master` branch is one behind `origin/master`. This is similar to what we mentioned at the beginning of the chapter because we are removing and undoing commits that are already published. Also, as mentioned, you should *only* perform the undo and redo (`git reset`) operations on commits that are not shared yet. Here, we only show it on the published commits to make the example easy to reproduce.

How it works...

Effectively, we are just changing the pointer of the `master` branch to point to the previous commit `HEAD`, which means the first parent of

HEAD. Now the branch will point to `9c7532f`, instead of the commit we removed, `35b29ae`. This is shown in the following figure:



The figure also shows that the original `3061dc6` commit is still present in the repository, but new commits on the `master` branch will start from `9c7532f`; the `3061dc6` commit is called a **dangling** commit.

Tip

You should only do this undo operation to commits you haven't shared (pushed) yet, as when you create new commits after undo or reset, those commits form a new history that will diverge from the original history of the repository.

When the `reset` command is executed, Git looks at the commit pointed to by `HEAD` and finds the parent commit from this. The current branch, `master`, and the `HEAD` pointer are then reset to the parent commit and so are the staging area and working tree.

Undo – remove a commit and retain the changes to files

Instead of performing the hard reset and thereby losing all the changes the commit introduced, the reset can be performed so that the changes are retained in the working directory.

Getting ready

We'll again use the example of the `hello world` repository. Make a fresh clone of the repository, or reset the `master` branch if you have already cloned one.

You can make a fresh clone as follows:

```
$ git clone https://github.com/dvaske/hello_world_cookbook.git  
$ cd hello_world_cookbook
```

You can reset the existing clone as follows:

```
$ cd hello_world_cookbook  
$ git checkout master  
$ git reset --hard origin master  
HEAD is now at 3061dc6 Adds Java version of 'hello world'
```

How to do it...

1. First, we'll check whether we have made no changes to files in the working tree (just for the clarity of the example) and the history of the repository:

```
$ git status  
On branch master  
Your branch is up-to-date with 'origin/master'.  
  
nothing to commit, working directory clean
```

```
$ git log --oneline  
3061dc6 Adds Java version of 'hello world'  
9c7532f Fixes compiler warnings
```

5b5d692 Initial commit, K&R hello world

- Now, we'll undo the commit and retain the changes introduced in the working tree:

```
$ git reset --mixed HEAD^
```

```
$ git log --oneline  
9c7532f Fixes compiler warnings  
5b5d692 Initial commit, K&R hello world
```

```
$ 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)
```

Untracked files:

```
(use "git add <file>..." to include in what will be  
committed)
```

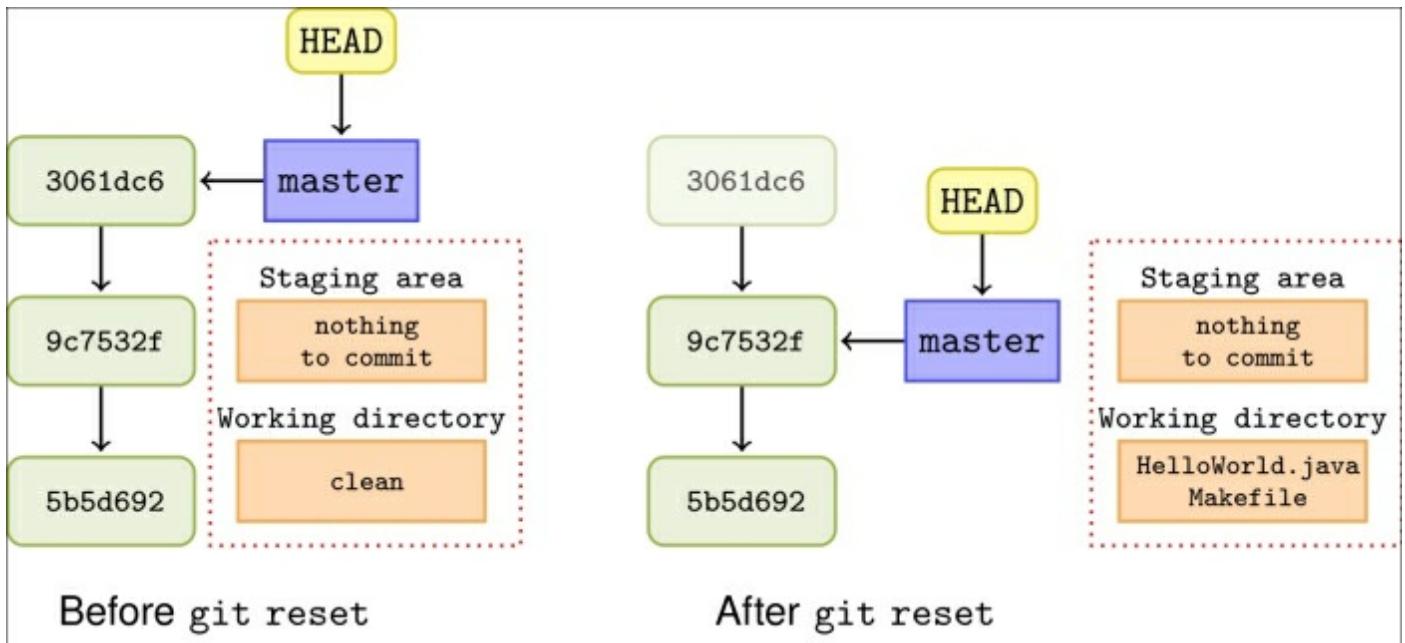
```
HelloWorld.java  
Makefile
```

```
nothing added to commit but untracked files present (use  
"git add" to track)
```

We can see that our commit is undone, but the changes to the file are preserved in the working tree, so more work can be done in order to create a proper commit.

How it works...

From the parent commit pointed to by the commit at `HEAD`, Git resets the branch pointer and `HEAD` to point to the parent commit. The staging area is reset, but the working tree is kept as it was before the reset, so the files affected by the undone commit will be in the modified state. The following figure depicts this:



Tip

The `--mixed` option is the default behavior of `git reset`, so it can be omitted:

```
git reset HEAD^
```

Undo – remove a commit and retain the changes in the staging area

Of course, it is also possible to undo the commit, but keep the changes to the files in the index or the staging area so that you are ready to recreate the commit with, for example, some minor modifications.

Getting ready

We'll still use the example of the `hello world` repository. Make a fresh clone of the repository, or reset the `master` branch if you have already cloned one.

Create a fresh clone as follows:

```
$ git clone https://github.com/dvaske/hello_world_cookbook.git  
$ cd hello_world_cookbook
```

We can reset the existing clone as follows:

```
$ cd hello_world_cookbook  
$ git checkout master  
$ git reset --hard origin master  
HEAD is now at 3061dc6 Adds Java version of 'hello world'
```

How to do it...

1. Check whether we have no files in the modified state and check the log:

```
$ git status  
On branch master  
Your branch is up-to-date with 'origin/master'.  
  
nothing to commit, working directory clean  
  
$ git log --oneline
```

```
3061dc6 Adds Java version of 'hello world'  
9c7532f Fixes compiler warnings  
5b5d692 Initial commit, K&R hello world
```

2. Now, we can undo the commit, while retaining the changes in the index:

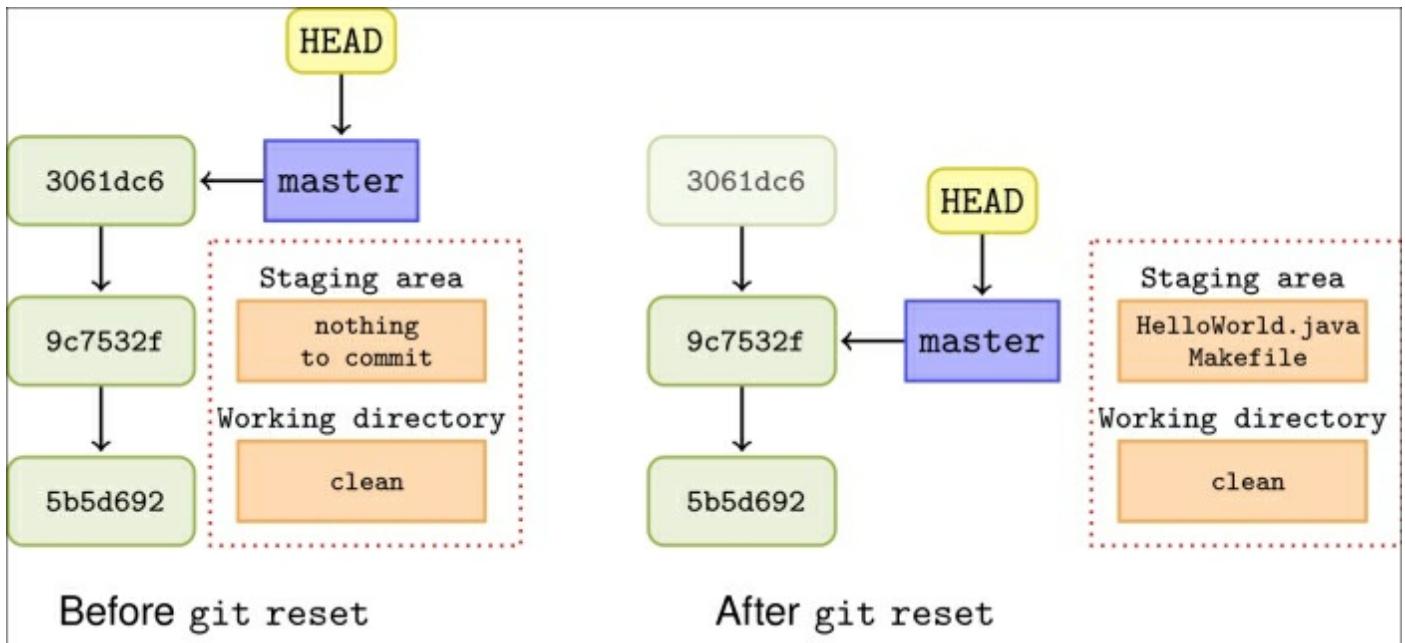
```
$ git reset --soft HEAD^  
  
$ git log --oneline  
9c7532f Fixes compiler warnings  
5b5d692 Initial commit, K&R hello world  
  
$ git status  
On branch master  
Your branch is behind 'origin/master' by 1 commit, and can  
be fast-forwarded.  
(use "git pull" to update your local branch)  
  
Changes to be committed:  
(use "git reset HEAD <file>..." to unstage)  
  
  new file:   HelloWorld.java  
  new file:   Makefile
```

You can now do minor (or major) changes to the files you need, add them to the staging area, and create a new commit.

How it works...

Again, Git will reset the branch pointer and `HEAD` to point to the previous commit. However, with the `--soft` option, the index and working directories are not reset, that is, they have the same state just as they had before we created the now undone commit.

The following figure shows the Git state before and after the undo:



Undo – working with a dirty area

In the previous examples, we assumed that the working tree was clean, that is, no tracked files are in the modified state. However, this is not always the case, and if a *hard reset* is done, the changes to the modified files will be lost. Fortunately, Git provides a smart way to quickly put stuff away so that it can be retrieved later using the `git stash` command.

Getting ready

Again, we'll use the example of the `hello world` repository. Make a fresh clone of the repository, or reset the `master` branch if you have already cloned.

We can create the fresh clone as follows:

```
$ git clone https://github.com/dvaske/hello_world_cookbook.git
$ cd hello_world_cookbook
```

We can reset the existing clone as follows:

```
$ cd hello_world_cookbook
$ git checkout master
$ git reset --hard origin master
HEAD is now at 3061dc6 Adds Java version of 'hello world'
```

We'll also need to have some files in the working condition, so we'll change `hello_world.c` to the following:

```
#include <stdio.h>

void say_hello(void) {
    printf("hello, world\n");
}

int main(void) {
    say_hello();
    return 0;
}
```

How to do it...

In order not to accidentally delete any changes you have in your working tree when you are about to undo a commit, you can have a look at the current state of your working directory with `git status` (as we already saw). If you have changes and you want to keep them, you can stash them away before undoing the commit and retrieve them afterwards. Git provides a `stash` command that can put unfinished changes away so it is easy to make quick context switches without losing work. The `stash` functionality is described further in [Chapter 12, Tips and Tricks](#). For now, you can think of the `stash` command like a stack where you can put your changes away and pop them later.

With the `hello_world.c` file in the working directory modified to the preceding state, we can try to do a hard reset on the `HEAD` commit, keeping our changes to the file by stashing them away before the reset and applying them back later:

1. First, check the history:

```
$ git log --oneline
3061dc6 Adds Java version of 'hello world'
9c7532f Fixes compiler warnings
5b5d692 Initial commit, K&R hello world
```

2. Then, check the status:

```
$ 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:   hello_world.c

no changes added to commit (use "git add" and/or "git
commit -a")
```

- As expected, `hello_world.c` was in the modified state; so, stash it away, check the status, and perform the reset:

```
$ git stash  
Saved working directory and index state WIP on master:  
3061dc6 Adds Java version of 'hello world'  
HEAD is now at 3061dc6 Adds Java version of 'hello world'  
  
$ git status  
On branch master  
Your branch is up-to-date with 'origin/master'.  
  
nothing to commit, working directory clean
```

```
$ git reset --hard HEAD^  
HEAD is now at 9c7532f Fixes compiler warnings  
$ git log --oneline  
9c7532f Fixes compiler warnings  
5b5d692 Initial commit, K&R hello world
```

- The reset is done, and we got rid of the commit we wanted. Let's resurrect the changes we stashed away and check the file:

```
$ git stash pop  
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)  
  
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:   hello_world.c  
  
no changes added to commit (use "git add" and/or "git  
commit -a")  
Dropped refs/stash@{0}  
(e56b68a1f5a0f72afcfd064ec13eefcda7a175ca)  
  
$ cat hello_world.c  
#include <stdio.h>  
  
void say_hello(void) {
```

```
    printf("hello, world\n");
}

int main(void) {
    say_hello();
    return 0;
}
```

So, the file is back to the state before the reset, and we got rid of the unwanted commit.

How it works...

The `reset` command works as explained in the previous examples, but combined with the `stash` command, it forms a very useful tool that corrects mistakes even though you have already starting working on something else. The `stash` command works by saving the current state of your working directory and the staging area. Then, it reverts your working directory to a clean state.

See also

- Refer to [Chapter 12, *Tips and Tricks*](#), for more details on the `git stash` command

Redo – recreate the latest commit with new changes

As with undo, redo can mean a lot of things. In this context, redoing a commit will mean creating almost the same commit again with the same parent(s) as the previous commit, but with different content and/or different commit messages. This is quite useful if you've just created a commit but perhaps have forgotten to add a necessary file to the staging area before you committed, or if you need to reword the commit message.

Getting ready

Again, we'll use the `hello world` repository. Make a fresh clone of the repository, or reset the `master` branch if you have already cloned.

We can create a fresh clone as follows:

```
$ git clone https://github.com/dvaske/hello_world_cookbook.git  
$ cd hello_world_cookbook
```

We can reset an existing clone as follows:

```
$ cd hello_world_cookbook  
$ git checkout master  
$ git reset --hard origin master  
HEAD is now at 3061dc6 Adds Java version of 'hello world'
```

How to do it...

Let's pretend we need to redo the latest commit because we need to reword the commit message to include a reference to the issue tracker.

1. Let's first take a look at the latest commit and make sure the working directory is clean:

```
$ git log -1  
commit 3061dc6cf7aeb2f8cb3dee651290bfea85cb4392  
Author: Aske Olsson <aske.olsson@switch-gears.dk>
```

Date: Sun Mar 9 14:12:45 2014 +0100

Adds Java version of 'hello world'

Also includes a makefile

```
$ git status
```

On branch master

Your branch is up-to-date with 'origin/master'.

nothing to commit, working directory clean

- Now, we can redo the commit and update the commit message with the `git commit --amend` command. This will bring up the default editor, and we can add a reference to the issue tracker in the commit message (`Fixes: RD-31415`):

```
$ git commit --amend
```

Adds Java version of 'hello world'

Also includes a makefile

Fixes: RD-31415

```
# Please enter the commit message for your changes. Lines
starting
```

```
# with '#' will be ignored, and an empty message aborts the
commit.
```

```
#
```

```
# Author: Aske Olsson <aske.olsson@switch-gears.dk>
```

```
#
```

```
# On branch master
```

```
# Your branch is up-to-date with 'origin/master'.
```

```
#
```

```
# Changes to be committed:
```

```
#       new file: HelloWorld.java
```

```
#       new file: Makefile
```

```
#
```

```
~
```

```
~
```

```
[master 75a41a2] Adds Java version of 'hello world'
```

```
Author: Aske Olsson <aske.olsson@switch-gears.dk>
```

```
2 files changed, 19 insertions(+)
```

```
create mode 100644 HelloWorld.java
```

```
create mode 100644 Makefile
```

- Now let's check the log again to see whether everything worked:

```
$ git log -1
commit 75a41a2f550325234a2f5f3ba41d35867910c09c
Author: Aske Olsson <aske.olsson@switch-gears.dk>
Date:   Sun Mar 9 14:12:45 2014 +0100
```

Adds Java version of 'hello world'

Also includes a makefile

Fixes: RD-31415

4. We can see that the commit message has changed, but we can't verify from the log output that the parent of the commit is the same as in the original commit, and so on, as we saw in the first commit we did. To check this, we can use the `git cat-file` command we learned about in [Chapter 1, Navigating Git](#). First, let's see how the original commit looked:

```
$ git cat-file -p 3061dc6
tree d3abe70c50450a4d6d70f391fcbd1a4609d151f
parent 9c7532f5e788b8805ffd419fcf2a071c78493b23
author Aske Olsson <aske.olsson@switch-gears.dk> 1394370765
+0100
committer Aske Olsson <aske@schantz.com> 1394569447 +0100
Adds Java version of 'hello world'
```

Also includes a makefile

The parent commit is b8c39bb35c4c0b00b6cfb4e0f27354279fb28866 and the root tree is d3abe70c50450a4d6d70f391fcbd1a4609d151f.

5. Let's check the data from the new commit:

```
$ git cat-file -p HEAD
tree d3abe70c50450a4d6d70f391fcbd1a4609d151f
parent 9c7532f5e788b8805ffd419fcf2a071c78493b23
author Aske Olsson <aske.olsson@switch-gears.dk> 1394370765
+0100
committer Aske Olsson <aske@schantz.com> 1394655225 +0100

Adds Java version of 'hello world'
```

Also includes a makefile

Fixes: RD-31415

The parent is the same, that is, 9c7532f5e788b8805ffd419fcf2a071c78493b23 and the root tree is also the same, that is, d3abe70c50450a4d6d70f391fcbd1a4609d151f. This is what we expected as we only changed the commit message. If we had added some changes to the staging area and executed `git commit--amend`, we would have included those changes in the commit and the root-tree SHA1 ID would have been different, but the parent commit ID still the same.

How it works...

The `--amend` option to `git commit` is roughly equivalent to performing `git reset -soft HEAD^`, followed by fixing the files needed and adding those to the staging area. Then, we will run `git commit` reusing the commit message from the previous commit (`git commit -c ORIG_HEAD`).

There is more...

We can also use the `--amend` method to add missing files to our latest commit. Let's say you needed to add the `README.md` file to your latest commit in order to get the documentation up to date, but you have already created the commit though you have not pushed it yet.

You then add the file to the index as you would while starting to craft a new commit. You can check with `git status` that only the `README.md` file is added:

```
$ git add README.md
$ git status
On branch master
Your branch and 'origin/master' have diverged,
and have 1 and 1 different commit each, respectively.
(use "git pull" to merge the remote branch into yours)
```

Changes to be committed:
`(use "git reset HEAD <file>..." to unstage)`

new file: README.md

Now you can amend the latest commit with `git commit --amend`. The command will include files in the index in the new commit and you can, as with the last example, reword the commit message if needed. It is not needed in this example, so we'll pass the `--no-edit` option to the command:

```
$ git commit --amend --no-edit
[master f09457e] Adds Java version of 'hello world'
Author: Aske Olsson <aske.olsson@switch-gears.dk>
3 files changed, 20 insertions(+)
create mode 100644 HelloWorld.java
create mode 100644 Makefile
create mode 100644 README.md
```

You can see from the output of the commit command that three files were changed and `README.md` was one of them.

Tip

You can also reset the author information (name, e-mail, and timestamp) with the `commit --amend` command. Just pass along the `--reset-author` option and Git will create a new timestamp and read author information from the configuration or environment, instead of the using the information from the old commit object.

Revert – undo the changes introduced by a commit

Revert can be used to undo a commit in history that has already been published (pushed), whereas this can't be done with the `amend` or `reset` options without rewriting history.

Revert works by applying the anti-patch introduced by the commit in question. A revert will, by default, create a new commit in history with a commit message that describes which commit has been reverted.

Getting ready

Again, we'll use the `hello world` repository. Make a fresh clone of the repository, or reset the `master` branch if you have already cloned.

We can create a fresh clone as follows:

```
$ git clone https://github.com/dvaske/hello_world_cookbook.git  
$ cd hello_world_cookbook
```

We can reset the existing clone as follows:

```
$ cd hello_world_cookbook  
$ git checkout master  
$ git reset --hard origin master  
HEAD is now at 3061dc6 Adds Java version of 'hello world'
```

How to do it...

1. First, we'll list the commits in the repository:

```
$ git log --oneline  
3061dc6 Adds Java version of 'hello world'  
9c7532f Fixes compiler warnings  
5b5d692 Initial commit, K&R hello world
```

2. We'll revert the second commit, `b8c39bb`:

```
$ git revert 9c7532f
```

```
Revert "Fixes compiler wanings"

This reverts commit
9c7532f5e788b8805ffd419fcf2a071c78493b23.

# Please enter the commit message for your changes. Lines
starting
# with '#' will be ignored, and an empty message aborts the
commit.
# On branch master
# Your branch is up-to-date with 'origin/master'.
#
# Changes to be committed:
#       modified:   hello_world.c
#
~
~
~
"~/aske/packt/repos/hello_world_cookbook/.git/COMMIT_EDITMSG"
G" 12L, 359C
[master 9b94515] Revert "Fixes compiler warnings"
 1 file changed, 1 insertion(+), 5 deletions(-)
```

3. When we check the log, we can see that a new commit has been made:

```
$ git log --oneline
9b94515 Revert "Fixes compiler warnings"
3061dc6 Adds Java version of 'hello world'
9c7532f Fixes compiler warnings
5b5d692 Initial commit, K&R hello world
```

We can take a closer look at the two commits with `git show` if we want a closer investigation of what happened.

How it works...

Git revert applies the anti-patch of the commit in question to the current `HEAD` pointer. It will generate a new commit with the "anti-patch" and a commit message that describes the reverted commit(s).

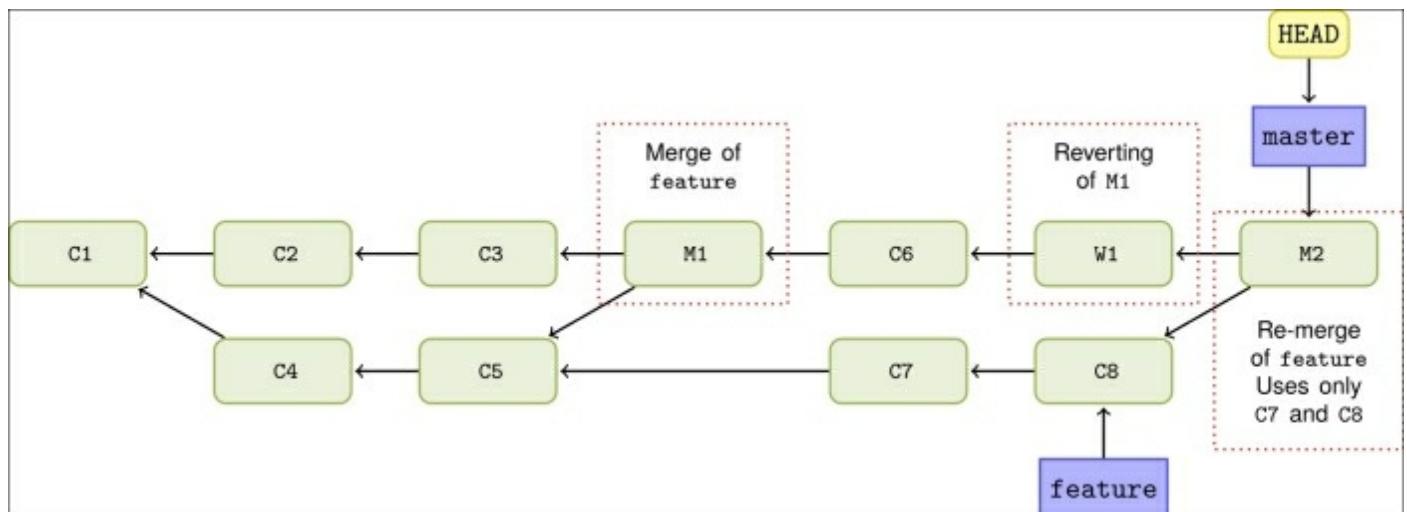
There's more...

It's possible to revert more than one commit in a single revert, for example, `git revert master~6..master~2` will revert the commits from the sixth commit from the bottom in `master` to the third commit from the bottom in `master` (both included).

It is also possible not to create a commit while reverting; passing the `-n` option to `git revert` will apply the needed patched but only to the working tree and the staging area.

Reverting a merge

Merge commits are a special case when it comes to revert. In order to be able to revert a merge commit, you'll have to specify which parent side of the merge you want to keep. However, when you revert a merge commit, you should keep in mind that though reverting will undo the changes to files, it doesn't undo history. This means that when you revert a merge commit, you declare that you will not have any of the changes introduced by the merge in the target branch. The effect of this is that the subsequent merges from the other branch will only bring in changes of commits that are not ancestors of the reverted merge commit.



In this example, we will learn how to revert a merge commit, and we'll learn how we can merge the branch again, getting all the changes merged by reverting the reverted merge commit.

Getting ready

Again, we'll use the `hello world` repository. Make a fresh clone of the repository, or reset the `master` branch if you have already cloned.

We can create a fresh clone as follows:

```
$ git clone https://github.com/dvaske/hello_world_cookbook.git
$ cd hello_world_cookbook
```

We can reset the existing clone as follows:

```
$ cd hello_world_cookbook
$ git checkout master
$ git reset --hard origin master
HEAD is now at 3061dc6 Adds Java version of 'hello world'
```

In this example, we also need to use some of the other branches in the repository, so we need to create them locally:

```
$ git branch -f feature/p-lang origin/feature/p-lang
Branch feature/p-lang set up to track remote branch feature/p-
lang from origin.
$ git checkout develop
Switched to branch 'develop'
Your branch is up-to-date with 'origin/develop'.
$ git reset --hard origin/develop
HEAD is now at a95abc6 Adds Groovy hello world
```

How to do it...

On the `develop` branch, we have just checked out that there is a merge commit that introduces "hello world" programs from languages that start with P. Unfortunately, the Perl version doesn't run:

```
$ perl hello_world.pl
Can't find string terminator '"' anywhere before EOF at
hello_world.pl line 3.
```

The following steps will help you revert a merge:

1. Let's take a look at history, the latest five commits, and find the merge commit:

```
$ git log --oneline --graph -5
* a95abc6 Adds Groovy hello world
* 5ae3beb Merge branch 'feature/p-lang' into develop
|\ 
| * 7b29bc3 php version added
| * 9944417 Adds perl hello_world script
* | ed9af38 Hello world shell script
|/
```

The commit we are looking for is `5ae3beb Merge branch`

'feature/p-lang' into develop; this adds the commits for "hello world" in Perl and PHP to the `develop` branch. We would like the fix of the Perl version to happen on the `feature` branch, and then merge it to `develop` when ready. In order to keep `develop` stable, we need to revert the merge commit that introduced the faulty Perl version. Before we perform the merge, let's just have a look at the contents of `HEAD`:

```
$ git ls-tree --abbrev HEAD
100644 blob 28f40d8    helloWorld.groovy
100644 blob 881ef55    hello_world.c
100644 blob 5dd01c1    hello_world.php
100755 blob ae06973   hello_world.pl
100755 blob f3d7a14   hello_world.py
100755 blob 9f3f770   hello_world.sh
```

2. Revert the merge, keeping the history of the first parent:

```
$ git revert -m 1 5ae3beb
[develop e043b95] Revert "Merge branch 'feature/p-lang'
into develop"
 2 files changed, 4 deletions (-)
 delete mode 100644 hello_world.php
 delete mode 100755 hello_world.pl
```

3. Let's have a look at the contents of our new `HEAD` state:

```
$ git ls-tree --abbrev HEAD
100644 blob 28f40d8    helloWorld.groovy
100644 blob 881ef55    hello_world.c
100755 blob f3d7a14   hello_world.py
100755 blob 9f3f770   hello_world.sh
```

The Perl and PHP files introduced in the merge are gone, so the revert did its job.

How it works...

The `revert` command will take the patches introduced by the commit you want to revert and apply the reverse/anti patch to the working tree. If all goes well, that is, there are no conflicts, a new commit will be made. While reverting a merge commit, only the changes introduced in

the mainline (the `-m` option) will be kept, and all the changes introduced in the other side of the merge will be reverted.

There is more...

Though it is easy to revert a merge commit, you might run into issues if you later want to the branch again because the issues on the merge have not been fixed. When revert of the merge commit is performed, you actually tell Git that you do not want any of the changes that the other branch introduced in this branch. So, when you try to merge in the branch again, you will only get the changes from the commits that are not ancestors of the reverted merge commit.

We will see this in action by trying to merge the `feature/p-lang` branch with the `develop` branch again:

```
$ git merge --no-edit feature/p-lang
CONFLICT (modify/delete): hello_world.pl deleted in HEAD and
modified in feature/p-lang. Version feature/p-lang of
hello_world.pl left in tree.
Automatic merge failed; fix conflicts and then commit the
result.
```

We can solve the conflict just by adding `hello_world.pl`:

```
$ git add hello_world.pl
$ git commit
[develop 2804731] Merge branch 'feature/p-lang' into develop
```

Let's check the tree if everything seems right:

```
$ git ls-tree --abbrev HEAD
100644 blob 28f40d8    helloWorld.groovy
100644 blob 881ef55    hello_world.c
100755 blob 6611b8e    hello_world.pl
100755 blob f3d7a14    hello_world.py
100755 blob 9f3f770    hello_world.sh
```

The `hello_world.php` file is missing, but this makes sense as the change that introduced it was reverted in the reverted merge commit.

To perform a proper re-merge, we first have to revert the reverting merge commit; this can seem a bit weird, but it is the way to get the changes from before the revert back into our tree. Then, we can perform another merge of the branch, and we'll end up with all the changes introduced by the branch we're merging in. However, we first we have to discard the merge commit we just made with a hard reset:

```
$ git reset --hard HEAD^
HEAD is now at c46deed Revert "Merge branch 'feature/p-lang'
into develop"
```

Now, we can revert the reverting merge and re-merge the branch:

```
$ git revert HEAD
[develop 9950c9e] Revert "Revert "Revert "Merge branch 'feature/p-lang'
into develop"""
2 files changed, 4 insertions(+)
create mode 100644 hello_world.php
create mode 100755 hello_world.pl

$ git merge feature/p-lang
Merge made by the 'recursive' strategy.
 hello_world.pl | 2 ++
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Let's check the tree for the Perl and PHP files, and see whether the Perl file has been fixed:

```
$ git ls-tree --abbrev HEAD
100644 blob 28f40d8    helloWorld.groovy
100644 blob 881ef55    hello_world.c
100644 blob 5dd01c1    hello_world.php
100755 blob 6611b8e    hello_world.pl
100755 blob f3d7a14    hello_world.py
100755 blob 9f3f770    hello_world.sh

$ perl hello_world.pl
Hello, world!
```

See also

For more information on reverting merges, refer to the following articles:

- The *How To Revert a Faulty Merge* article at
<https://www.kernel.org/pub/software/scm/git/docs/howto/revert-a-faulty-merge.html>
- The *Undoing Merges* article at <http://git-scm.com/blog/2010/03/02/undoing-merges.html>

Viewing past Git actions with git reflog

The `reflog` command stores information on updates to the tip of the branches in Git, where the normal `git log` command shows the ancestry chain from `HEAD`, and the `reflog` command shows what `HEAD` has pointed to in the repository. This is your history in the repository that tells how you have moved between branches, created your commits and resets, and so on. Basically, anything that makes `HEAD` point to something new is recorded in the `reflog`. This means that by going through `reflog` command, you can find *lost* commits that none of your branches nor other commits point to. This makes the `reflog` command a good starting point for trying to find a lost commit.

Getting ready

Again, we'll use the `hello world` repository. If you make a fresh clone, make sure to run the scripts for this chapter so that there will be some entries in the `reflog` command. The scripts can be found on the book's homepage. If you just reset the `master` branch to `origin/master` after performing the recipes in this chapter, everything is ready.

We can create a fresh clone as follows:

```
$ git clone https://github.com/dvaske/hello_world_cookbook.git  
$ cd hello_world_cookbook
```

We can reset an existing clone as follows:

```
$ cd hello_world_cookbook  
$ git checkout master  
$ git reset --hard origin master  
HEAD is now at 3061dc6 Adds Java version of 'hello world'
```

How to do it...

1. Let's try to run the `reflog` command and limit ourselves to just the

latest seven entries:

```
$ git reflog -7
3061dc6 HEAD@{0}: checkout: moving from develop to master
d557284 HEAD@{1}: merge feature/p-lang: Merge made by the
'recursive' strategy.
9950c9e HEAD@{2}: revert: Revert "Revert "Merge branch
'feature/p-lang' into develop"""
c46deed HEAD@{3}: reset: moving to HEAD^
2804731 HEAD@{4}: commit (merge): Merge branch 'feature/p-
lang' into develop
c46deed HEAD@{5}: revert: Revert "Merge branch 'feature/p-
lang' into develop"
a95abc6 HEAD@{6}: checkout: moving from master to develop
```

Note

In your repository, the commits will have different SHA-1 hashes due to the fact that the commits generated in the examples will have slightly different content, specifically your user name and e-mail address, but the order should be approximately the same.

We can see the movements we did in the last example by reverting, committing, and resetting. We can see the merge commit, 2804731, that we abandoned. It didn't merge in all the changes we wanted it to due to the previous merge and its revert.

2. We can take a closer look at the commit with `git show`:

```
$ git show 2804731
commit 2804731c3abc4824cdab66dc7567bed4cddde0d3
Merge: c46deed 32fa2cd
Author: Aske Olsson <aske@schantz.com>
Date:   Thu Mar 13 23:20:21 2014 +0100

        Merge branch 'feature/p-lang' into develop

Conflicts:
    hello_world.pl
```

Indeed, this was the commit we chose to abandon in the previous example. We can also look at the tree of the commit, just as we did in the previous example, and check whether they are the same:

```
git ls-tree --abbrev 2804731
100644 blob 28f40d8    helloWorld.groovy
100644 blob 881ef55    hello_world.c
100755 blob 6611b8e    hello_world.pl
100755 blob f3d7a14    hello_world.py
100755 blob 9f3f770    hello_world.sh
```

From here, there are various ways to resurrect the changes. You can either checkout the commit and create a branch; then, you'll have a pointer so you can easily find it again. You can also checkout specific files from the commit with `git checkout - path/to/file SHA-1`, or you can use the `git show` or `git cat-file` commands to view the files.

How it works...

For every movement of the `HEAD` pointer in the repository, Git stores the commit pointed to and the action for getting there. This can be commit, checkout, reset, revert, merge, rebase, and so on. The information is local to the repository and is not shared on pushes, fetches, and clones. Using the `reflog` command to find the lost commits is fairly easy if you know what you are searching for and the approximate time when you created the commit you are searching for. If you have a lot of reflog history, many commits, switching branches, and so on, it can be hard to search through the `reflog` command due to the amount of noise from the many updates to `HEAD`. The output of the `reflog` command can be a lot of options, and among them, there are options you can also pass on to the normal `git log` command.

Finding lost changes with git fsck

Another tool exists in Git that can help you find and recover lost commits and even blobs (files), which is `git fsck`. The `fsck` command tests the object database and verifies the SHA-1 ID of the objects and the connections they make. The command can also be used to find objects that are not reachable from any named reference, as it tests all the objects found in the database, which are under the `.git/objects` folder.

Getting ready

Again, we'll use the `hello world` repository. If you make a fresh clone, make sure to run the scripts for this chapter (`04_undo_dirty.sh`), so there will be some objects for `git fsck` to consider. The scripts can be found on the book's homepage. If you just reset the `master` branch after performing the other recipes in the chapter, everything is ready.

We can create the fresh clone as follows:

```
$ git clone https://github.com/dvaske/hello_world_cookbook.git  
$ cd hello_world_cookbook
```

We can reset an existing clone as follows:

```
$ cd hello_world_cookbook  
$ git checkout master  
$ git reset --hard origin master  
HEAD is now at 3061dc6 Adds Java version of 'hello world'
```

How to do it...

1. Let's look for the unreachable objects in the database:

```
$ git fsck --unreachable  
Checking object directories: 100% (256/256), done.  
unreachable commit 147240ad0297f85c9ca3ed513906d4b75209e83d  
unreachable blob b16cf63ab66605f9505c17c5affd88b34c9150ce  
unreachable commit 4c3b1e10d8876cd507bcf2072c85cc474f7fb93b
```

Note

The object's ID, the SHA-1 hash, will not be the same if you perform the example on your computer, as the committer, author, and timestamp will be different.

2. We found two commits and one blob. Let's take a closer look at each of them; the blob first:

```
git show b16cf63ab66605f9505c17c5affd88b34c9150ce
#include <stdio.h>

void say_hello(void) {
    printf("hello, world\n");
}

int main(void) {
    say_hello();
    return 0;
}
```

So the blob is the `hello_world.c` file from the example which stashing away your changes before resetting a commit. Here we stashed away the file, performed a reset, and resurrected the file from the stash, but we never actually performed a commit. The `stash` command, however, did add the file to the database, so it could find it again, and the file will continue to be there until the garbage collection kicks in or forever if it is referenced by a commit in the general history.

3. Let's look closer at the two commits:

```
$ git show 147240ad0297f85c9ca3ed513906d4b75209e83d
commit 147240ad0297f85c9ca3ed513906d4b75209e83d
Merge: 3061dc6 4c3b1e1
Author: Aske Olsson <aske@schantz.com>
Date:   Thu Mar 13 23:19:37 2014 +0100

    WIP on master: 3061dc6 Adds Java version of 'hello
world'

diff --cc hello_world.c
index 881ef55,881ef55..b16cf63
```

```

--- a/hello_world.c
+++ b/hello_world.c
@@@ -1,7 -1,7 +1,10 @@@
 #include <stdio.h>

--int main(void) {
++void say_hello(void) {
    printf("hello, world\n");
++}

++int main(void) {
++  say_hello();
    return 0;
--}
++}

$ git show 4c3b1e10d8876cd507bcf2072c85cc474f7fb93b
commit 4c3b1e10d8876cd507bcf2072c85cc474f7fb93b
Author: Aske Olsson <aske@schantz.com>
Date:   Thu Mar 13 23:19:37 2014 +0100

    index on master: 3061dc6 Adds Java version of 'hello
world'

```

Both the commits are actually commits we made when we stashed away our changes in the previous example. The `stash` command creates a commit object with the contents of the staging area, and a merge commit merging `HEAD` and the commit with the index with the content of the working directory (tracked files only). As we resurrected our stashed changes in the previous example, we no longer have any reference pointing at the preceding commits; therefore, they are found by `git fsck`.

How it works...

The `git fsck` command will test all the objects found under the `.git/objects` folder. When the `--unreachable` option is given, it will report the objects found that can't be reached from another reference; the reference can be a branch, a tag, a commit, a tree, the reflog, or stashed away changes.

See also

- Refer to [Chapter 12, *Tips and Tricks*](#), for more information on the
`git stash` command

Chapter 9. Repository Maintenance

In this chapter, we will cover the following topics:

- Pruning remote branches
- Running garbage collection manually
- Turning off automatic garbage collection
- Splitting a repository
- Rewriting history – changing a single file
- Creating a backup of your repositories as mirror repositories
- A quick submodule how-to
- Subtree merging
- Submodule versus subtree merging

Introduction

In this chapter, we'll take a look at the various tools for repository maintenance. We will look at how we can easily delete branches in the local repository that have been deleted from the remote repository. We'll also see how we can trigger garbage collection and how to turn it off. We will take a look at how a repository can be split with the `filter-branch` command and how the `filter-branch` command can be used to rewrite the history of a repository. Finally, we'll take a quick look on how to integrate other git projects as subprojects in a git repository with either the submodule functionality or the subtree strategy.

Pruning remote branches

Often, development in a software project tracked with Git happens on feature branches, and as time goes by, an increasing number of feature branches are merged to the mainline. Usually, these feature branches are deleted in the `main` repository (`origin`). However, the branches are not automatically deleted from all the clones while fetching and pulling. Git must explicitly be told to delete the branches from the local repository that have been deleted on `origin`.

Getting ready

First, we'll set up two repositories and use one of them as a remote for the other. We will use the `hello_world_flow_model` repository, but first we'll clone a repository to a local `bare` repository:

```
$ git clone --bare  
https://github.com/dvaske/hello_world_flow_model.git  
hello_world_flow_model_remote
```

Next, we'll clone the newly cloned repository to a local one with a working directory:

```
$ git clone hello_world_flow_model_remote  
hello_world_flow_model
```

Now, let's delete a couple of merged feature branches in the `bare` repository:

```
$ cd hello_world_flow_model_remote  
$ git branch -D feature/continents  
$ git branch -D feature/printing  
$ git branch -D release/1.0  
$ cd ..
```

Finally, change the directory to your working copy and make sure `develop` is checked out:

```
$ cd hello_world_flow_model  
$ git checkout develop
```

```
$ git reset --hard origin/develop
```

How to do it...

1. Start by listing all branches using the following command:

```
$ git branch -a
* develop
  remotes/origin/HEAD -> origin/develop
  remotes/origin/develop
  remotes/origin/feature/cities
  remotes/origin/feature/continents
  remotes/origin/feature/printing
  remotes/origin/master
  remotes/origin/release/1.0
```

2. Let's try to fetch or pull and see whether anything happens using the following command:

```
$ git fetch
$ git pull
Current branch develop is up to date.
$ git branch -a
* develop
  remotes/origin/HEAD -> origin/develop
  remotes/origin/develop
  remotes/origin/feature/cities
  remotes/origin/feature/continents
  remotes/origin/feature/printing
  remotes/origin/master
  remotes/origin/release/1.0
```

3. The branches are still there even if they are deleted in the remote repository. We need to tell Git explicitly to delete the branches that are also deleted from the remote repository using the following command:

```
$ git fetch --prune
  x [deleted]          (none)      ->
origin/feature/continents
  x [deleted]          (none)      -> origin/feature/printing
  x [deleted]          (none)      -> origin/release/1.0
$ git branch -a
* develop
  remotes/origin/HEAD -> origin/develop
  remotes/origin/develop
```

```
remotes/origin/feature/cities  
remotes/origin/master
```

The branches are now also deleted from our local repository.

How it works...

Git simply checks the remote-tracking branches under the `remotes` or `origin` namespace and removes branches that are not found on the remote any more.

There's more...

There are several ways to remove the branches in Git that have been deleted on the master. It can be done while updating the local repository, as we saw with `git fetch --prune`, and also with `git pull --prune`. It can even be performed with the `git remote prune origin` command. This will also remove the branches from Git that are no longer available on the remote, but it will not update remote-tracking branches in the repository.

Running garbage collection manually

When using Git on a regular basis, you might notice that some commands sometimes trigger Git to perform garbage collection and pack loose objects into a pack file (Git's objects storage). The garbage collection and packing of loose objects can also be triggered manually by executing the `git gc` command. Triggering `git gc` is useful if you have a lot of loose objects. A loose object can, for example, be a blob or a tree or a commit. As we saw in [Chapter 1, Navigating Git](#), `blob`-, `tree`-, and `commit` objects are added to Git's database when we add files and create commits. These objects will first be stored as loose objects in Git's object storage as single files inside the `.git/objects` folder. Eventually, or by manual request, Git packs the loose objects into pack files that can reduce disk usage. A lot of loose objects can happen after adding a lot of files to Git, for example, when starting a new project or after frequent adds and commits. Running the garbage collection will make sure loose objects are being packed, and objects not referred to by any reference or object will be deleted. The latter is useful when you have deleted some branches/commits and want to make sure the objects referenced by these are also deleted.

Let's see how we can trigger garbage collection and remove some objects from the database.

Getting ready

First, we need a repository to perform the garbage collection on. We'll use the same repository as the previous example:

```
$ git clone  
https://github.com/dvaske/hello_world_flow_model.git  
$ cd hello_world_flow_model  
$ git checkout develop  
$ git reset --hard origin/develop
```

How to do it...

1. First, we'll check the repository for loose objects; we can do this with the `count-objects` command:

```
$ git count-objects  
51 objects, 204 kilobytes
```

2. We'll also check for unreachable objects, which are objects that can't be reached from any reference (tag, branch, or other object). The unreachable objects will be deleted when the garbage collect runs. We also check the size of the `.git` directory using the following command:

```
$ git fsck --unreachable  
Checking object directories: 100% (256/256), done.  
$ du -sh .git  
292K .git
```

3. There are no unreachable objects. This is because we just cloned and haven't actually worked in the repository. If we delete the `origin` remote, the remote branches (`remotes/origin/*`) will be deleted, and we'll lose the reference to some of the objects in the repository; they'll be displayed as unreachable while running `fsck` and can be garbage collected:

```
$ git remote rm origin  
$ git fsck --unreachable  
Checking object directories: 100% (256/256), done.  
unreachable commit 127c621039928c5d99e4221564091a5bf317dc27  
unreachable commit 472a3dd2fda0c15c9f7998a98f6140c4a3ce4816  
unreachable blob e26174ff5c0a3436454d0833f921943f0fc78070  
unreachable commit f336166c7812337b83f4e62c269deca8ccfa3675
```

4. We can see that we have some unreachable objects due to the deletion of the remote. Let's try to trigger garbage collection manually:

```
$ git gc  
Counting objects: 46, done.  
Delta compression using up to 8 threads.  
Compressing objects: 100% (44/44), done.  
Writing objects: 100% (46/46), done.  
Total 46 (delta 18), reused 0 (delta 0)
```

5. If we investigate the repository now, we can see the following:

```
$ git count-objects
5 objects, 20 kilobytes
$ git fsck --unreachable
Checking object directories: 100% (256/256), done.
Checking objects: 100% (46/46), done.
unreachable commit 127c621039928c5d99e4221564091a5bf317dc27
unreachable commit 472a3dd2fda0c15c9f7998a98f6140c4a3ce4816
unreachable blob e26174ff5c0a3436454d0833f921943f0fc78070
unreachable commit f336166c7812337b83f4e62c269deca8ccfa3675
$ du -sh .git
120K .git
```

6. The object count is smaller; Git packed the objects to the pack-file stored in the `.git/objects/pack` folder. The size of the repository is also smaller as Git compresses and optimizes the objects in the pack-file. However, there are still some unreachable objects left. This is because the objects will only be deleted if they are older than what is specified in the `gc.pruneexpire` configuration option that defaults to two weeks (config value: `2.weeks.ago`). We can override the default or configured option by running the `--prune=now` option:

```
$ git gc --prune=now
Counting objects: 46, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (26/26), done.
Writing objects: 100% (46/46), done.
Total 46 (delta 18), reused 46 (delta 18)
```

7. Investigating the repository gives the following output:

```
$ git count-objects
0 objects, 0 kilobytes
$ git fsck --unreachable
Checking object directories: 100% (256/256), done.
Checking objects: 100% (46/46), done.
$ du -sh .git
100K .git
```

The unreachable objects have been deleted, there are no loose objects, and the repository size is smaller now that the objects have been deleted.

How it works...

The `git gc` command optimizes the repository by compressing file revisions and deleting objects that are no longer referred to. The objects can be commits and so on. On an abandoned (deleted) branch, blobs from invocations of `git add`, commits discarded/redone with `git commit -amend`, or other commands that can leave objects behind. Objects are, by default, already compressed with zlib when they are created, and when moved into the pack-file, Git makes sure only to store the necessary change. If, for example, you change only a single line in a large file, it would waste a bit of space while storing the entire file in the pack-file again. Instead, Git stores the newest file as a whole in the pack-file and only the delta for the older version. This is pretty smart as you are more likely to require the newest version of the file, and Git doesn't have to do delta calculations for this. This might seem like a contradiction to the information from [Chapter 1, Navigating Git](#), where we learned that Git stores snapshots and not deltas. However, remember how the snapshot is made. Git hashes all the files content in blobs, makes tree and commit objects, and the commit object describes the full tree state with the root-tree `sha-1` hash. The storing of the objects inside the pack-files have no effect on the computation of the tree state. When you checkout an earlier version or commit, Git makes sure the `sha-1` hashes match the branch or commit or tag you requested.

Turning off automatic garbage collection

The automatic triggering of garbage collection can be turned off so it will not run unless manually triggered. This can be useful if you are searching the repository for a lost commit/file and want to make sure that it is not being garbage collected while searching (running Git commands).

Getting ready

We'll use the `hello_world_flow_model` repository again for this example:

```
$ git clone  
https://github.com/dvaske/hello_world_flow_model.git  
Cloning into 'hello_world_flow_model'...  
remote: Reusing existing pack: 51, done.  
remote: Total 51 (delta 0), reused 0 (delta 0)  
Unpacking objects: 100% (51/51), done.  
Checking connectivity... done.  
$ cd hello_world_flow_model  
$ git checkout develop  
Already on 'develop'  
Your branch is up-to-date with 'origin/develop'.  
$ git reset --hard origin/develop  
HEAD is now at 2269dcf Merge branch 'release/1.0' into develop
```

How to do it...

1. To switch off the automatic garbage collection triggering, we need to set the `gc.auto` configuration to 0. First, we'll check the existing setting, and then we can set it and verify the configuration using the following commands:

```
$ git config gc.auto  
$ git config gc.auto 0  
$ git config gc.auto  
0
```

2. Now we can try to run `git gc` with the `--auto` option, as it will be

called when normally triggered from other command:

```
$ git gc --auto
```

3. As expected, nothing happens as the configuration disables automatic garbage collection. We can still trigger it manually though (without the --auto flag):

```
$ git gc
Counting objects: 51, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (49/49), done.
Writing objects: 100% (51/51), done.
Total 51 (delta 23), reused 0 (delta 0)
```

Splitting a repository

Sometimes a project tracked with Git is not one logical project but several projects. This may be fully intentional and there is nothing wrong with it, but there can also be cases where the projects tracked in the same Git repository really should belong to two different repositories. You can imagine a project where the code base grows and at some point in time, one of the subprojects could have value as an independent project. This can be achieved by splitting the subfolders and/or files that contain the project that should have its own repository with the full history of commits touching the files and/or folders.

Getting ready

In this example, we'll use the JGit repository so we'll have some history to filter through. The subfolders we split out to are not really projects, but serve well as an example for this exercise.

1. First, clone the Jgit repository and create local branches of the remote ones using the following command:

```
git clone https://git.eclipse.org/r/jgit/jgit
Cloning into 'jgit'...
remote: Counting objects: 1757, done
remote: Finding sources: 100% (148/148)
remote: Total 46381 (delta 4), reused 46317 (delta 4)
Receiving objects: 100% (46381/46381), 10.86 MiB | 411.00
KiB/s, done.
Resolving deltas: 100% (24829/24829), done.
Checking connectivity... done.
$ cd jgit
$ git checkout master
Already on 'master'
Your branch is up-to-date with 'origin/master'.
```

2. Save the name of the current branch in the `current` variable:

```
$ current=$(git rev-parse --symbolic-full-name --abbrev-ref
HEAD)
```

3. In the following step, we create local branches from all the remote

branches in the repository:

```
$ for br in $(git branch -a | grep -v $current | grep remotes | grep -v HEAD);  
do  
    git branch ${br##*/} $br;  
done  
Branch stable-0.10 set up to track remote branch stable-0.10 from origin.  
Branch stable-0.11 set up to track remote branch stable-0.11 from origin.  
Branch stable-0.12 set up to track remote branch stable-0.12 from origin.  
...
```

First, we filtered the branches. From all the `git branch -a` branches, we exclude branches that match the `$current` variable somewhere in the name, `grep -v $current`. Then, we include only the branches that match `remote` with `grep remotes`. Finally, we exclude all branches with `HEAD` in `grep -v HEAD`. For each of the branches, `$br` we create a local branch with the name given after the last "/" in the full name of the branch: `git branch ${br##*/} $br`. For example, `remotes/origin/stable-0.10` becomes the local branch `stable-0.10`.

4. Now we'll prepare a small script that will delete everything but the input to the shell script from the Git index. Save the following to the file `clean-tree` in the folder that contains the Jgit repository (not the repository itself):

```
#!/bin/bash  
  
# Clean the tree for unwanted dirs and files  
# $1 Files and dirs to keep  
clean-tree () {  
    # Remove everything but $1 from the git index/staging  
    # area  
    for f in $(git ls-files | grep -E -v "$1" | grep -o -E  
    "^[^/]+" | sort -u); do  
        git rm -rq --cached --ignore-unmatch $f  
    done  
}  
  
clean-tree $1
```

The small script filters all the files currently in the staging area, `git ls-files`, excluding the ones that match the input `$1`, `grep -E -v "$1"`. It lists only the first part of their name/path up to the first `/`, `grep -o -E "^[^/\\"]+"`, and finally sorts them by unique entries using `sort -u`. The entries in the remaining list, `$f`, removed from the staging area of Git, `git rm -rq --cached --ignore-unmatch $f`. The `--cached` option tells Git to remove from the staging area and `--ignore-unmatched` tells Git not to fail if the file does not exist in the staging area. The `-rq` option is recursive and quiet respectively.

Tip

The staging area contains all the files tracked by Git in the last snapshot (commit) and files (modified or new) you have added with `git add`. However, you only see changes between the latest commit and the staging area when you run `git status`, and changes between the working tree and the staging area.

5. Make the file executable using the following command:

```
$ chmod +x clean-tree
```

6. Now we are ready to split out a subpart of the repository.

How to do it...

1. First, we'll decide which folders and files to keep in the new repository; we'll delete everything from the repository except those files. We'll store the files and folders to be kept in a string separated by `|` so that we can feed it to `grep` as a regular expression, as shown in the following command:

```
keep="org.eclipse.jgit.http|LICENSE|.gitignore|README.md|.gitattributes"
```

2. Now we are ready to start the conversion of the repository. We'll use the `git filter-branch` command that can rewrite the entire history of the repository; just what we need for this task.

Tip

Always remember to make sure you have a backup of the repository you are about to run `git filter-branch` on in case something goes wrong.

3. We'll use the `--index-filter` option to filter the branch. The option allows us to rewrite the index or staging area just before each commit is recorded, and we'll do this with the `clean-tree` script we created previously. We'll also preserve tags using `cat` as `tag-name-filter`. We will perform the rewrite on all branches. And remember to use the absolute path to the `clean-tree` script:

```
$ git filter-branch --prune-empty --index-filter  
"\"path/to/clean-tree\" \"$keep\""' --tag-name-filter cat --  
--all  
Rewrite 720734983bae056955bec3b36cc7e3847a0bb46a (13/3051)  
Rewrite 6e1571d5b9269ec79eadad0dbd5916508a4fee82 (23/3051)  
Rewrite 2bfe561f269afdd7f4772f8ebf34e5e25884942b (37/3051)  
Rewrite 2086fdaedd5e71621470865c34ad075d2668af99 (60/3051)  
...
```

4. The rewrite takes a bit of time as all commits need to be processed, and once the rewrite is done we can see that everything, except the files and folders we wanted to keep, is deleted.

```
git ls-tree --abbrev HEAD  
100644 blob f57840b .gitattributes  
100644 blob ea8c4bf .gitignore  
100644 blob 1b85c64 LICENSE  
100644 blob 6e6c0c7 README.md  
040000 tree 8bb062e org.eclipse.jgit.http.apache  
040000 tree 2dff82d org.eclipse.jgit.http.server  
040000 tree 91a9a3e org.eclipse.jgit.http.test
```

5. The cleanup isn't done just yet; Git filter-branch saves all the original references, branches and tags, under the `refs/original` namespace in the repository. After verifying, the new history looks good, and we can get rid of the original refs as these point to objects that are not in our current history, and take up a lot of disk space. We'll delete all the original refs and run the garbage collector to clean the repository for old objects:

```
$ du -sh .git  
17M .git
```

6. Delete original references, `refs/original`, and remove old objects with `git gc`, as shown in the following command:

```
$ git for-each-ref --format="% (refname)" refs/original/  
| \xargs -n 1 git update-ref -d  
$ git reflog expire --expire=now --all  
$ git gc --prune=now  
Counting objects: 3092, done.  
Delta compression using up to 8 threads.  
Compressing objects: 100% (1579/1579), done.  
Writing objects: 100% (3092/3092), done.  
Total 3092 (delta 1238), reused 1721 (delta 796)
```

7. Check the size of the repository after garbage collection:

```
$ du -sh .git  
796K .git
```

8. The repository is now clean of all old objects, the size is greatly reduced, and the history is preserved for the files and directories we listed to keep.

How it works...

The `git filter-branch` command has different filter options depending on what needs to be done when rewriting the repository. In this example, where we are only removing files and folders from the repository; the index-filter is highly useable as it allows us to rewrite the index just before recording a commit in the database without actually checking out the tree on disk, saving a lot of disk I/O. The clean-tree script we prepared previously is then used to remove the unwanted files and folders from the index. First, we list the contents of the index and filter the files and folders we want to keep. Then, we remove the remaining files and folders (`$f`) from the index with the following command:

```
git rm -rq --cached --ignore-unmatch $f
```

The `--cached` option tells Git to remove from files the index, and the `-rq` option tells to remove recursive (`r`) and be quiet (`q`). Finally, the `--ignore-unmatch` option is used so `git rm` will not exit with an error if it tries to remove a file that is not in the index.

There's more...

There are many more filters for `git filter-branch`; the most common ones and their use cases are:

- `env-filter`: This filter is used to modify the environment where commits are recorded. This is particularly useful when rewriting author and committer information.
- `tree-filter`: The tree-filter is used to rewrite the tree. This is useful if you need to add or modify files in the tree, for example, to remove sensitive data from a repository.
- `msg-filter`: This filter is used to update the commit message.
- `subdirectory-filter`: This filter can be used if you want to extract a single subdirectory to a new repository and keep the history of that subdirectory. The subdirectory will be the root of the new repository.

Rewriting history – changing a single file

In this example, we'll see how we can use Git filter-branch to remove sensitive data from a file throughout the repository history.

Getting ready

For simplicity, we'll use a very simple example repository. It contains a few files. One among them is `.credentials`, which contains a username and password. Start by cloning the repository and changing the directory, as shown in the following command:

```
$ git clone https://github.com/dvaske/remove-credentials.git  
$ cd remove-credentials
```

How to do it...

- As we need to modify a file when rewriting the history of this repository, we'll use the `tree-filter` option to filter branch. The `.credentials` file looks as follows:

```
username = foobar  
password = verysecret
```

- All we need to do is to remove everything after the equals sign on each line of the file. We can use the following `sed` command to do this:

```
sed -i '' 's/^(\.*=\').*\$/\1/'
```

- We can now run the filter branch with the following command:

```
$ git filter-branch --prune-empty --tree-filter "test -f .credentials && sed -i '' 's/^(\.*=\').*\$/\1/' .credentials || true" -- --all
```

- If we look at the file now, we can see the username and password are gone:

```
$ cat .credentials
```

```
username =  
password =
```

5. As we saw in the last example, we still need to clean up after the filter-branch, by deleting original references, expiring the `reflog`, and triggering garbage collection.

How it works...

For each commit in the repository, Git will check the contents of that commit and run `tree-filter`. If the filter fails, non zero the exit code, `filter-branch` will fail. Therefore, it is important to remember to handle the cases where `tree-filter` might fail. This is why the previous `tree-filter` checks whether the `.credentials` file exists, runs the `sed` command if it does, and otherwise returns `true` to continue the `filter-branch`.

Back up your repositories as mirror repositories

Though Git is distributed and every clone essentially is a backup, there are some tricks that can be useful when backing up Git repositories. A normal Git repository has a working copy of the files it tracks and the full history of the repository in the `.git` folder of that repository. The repositories on the server, the one you push to and pull from, will usually be `bare` repositories. A `bare` repository is a repository without a working copy. Roughly, it is just the `.git` folder of a normal repository. A `mirror` repository is almost the same as a `bare` repository, except it fetches all the references under `refs/*`, where a bare only fetches the references that fall under `refs/heads/*`. We'll now take a closer look at a normal, a `bare`, and a `mirror` clone of the Jgit repository.

Getting ready

We'll start by creating three clones of the Jgit repository, a normal, a `bare`, and a `mirror` clone. When we create the first clone, we can use that as a reference repository for the other clones. In this way, we can share the objects in the database, and we don't have to transfer the same data three times:

```
$ git clone https://git.eclipse.org/r/jgit/jgit
$ git clone --reference jgit --bare
https://git.eclipse.org/r/jgit/jgit
$ git clone --mirror --reference jgit
https://git.eclipse.org/r/jgit/jgit jgit.mirror
```

How to do it...

1. One of the differences between a normal repository and a `bare/mirror` one is that there are no remote branches in a `bare` repository; all the branches are created locally. We can see this in the three repositories by listing the branches with the `git branch` as follows:

```
$ cd jgit
$ git branch
* master

$ cd ../jgit.git # or cd ../jgit.mirror
$ git branch
* master
  stable-0.10
  stable-0.11
  stable-0.12
...
```

2. To see the difference between the `bare` and `mirror` repositories, we need to list the different `refs`pecs `fetch` and the different `refs` namespaces. List the `fetch` `refs`pec for `origin` in the mirror repository (`jgit.mirror`):

```
$ git config remote.origin.fetch
+refs/*:refs/*
```

3. List the different `refs` namespaces in the mirror repository:

```
$ git show-ref | cut -f2 -d " " | cut -f1,2 -d / | sort -u
refs/cache-automerge
refs/changes
refs/heads
refs/meta
refs/notes
refs/tags
```

4. There is no explicit `refs`pec `fetch` in the configuration for `origin` in the `bare` repository (`jgit.git`). When no configuration entry is found, Git uses the default `refs`pec `fetch`, as it does in a normal repository. We can check the remote URL of `origin` using the following command:

```
$ git config remote.origin.url
https://git.eclipse.org/r/jgit/jgit
```

5. List the different `refs` namespaces in the `bare` repository using the following command:

```
$ git show-ref | cut -f2 -d " " | cut -f1,2 -d / | sort -u
refs/heads
refs/tags
```

- Finally, we can list the `refs` fetch and `refs` namespaces for the normal repository (`jgit`):

```
$ git config remote.origin.fetch  
+refs/heads/*:refs/remotes/origin/*  
$ git show-ref | cut -f2 -d " " | cut -f1,2 -d / | sort -u  
refs/heads  
refs/remotes  
refs/tags
```

- The mirror repository has four ref namespaces not found in the normal or the bare repositories, `refs-` `cache-automerge`, `changes`, `meta`, and `notes`. The normal repository is the only one that has the `refs/remote` namespace.

How it works...

The normal and bare repositories are pretty similar, only the mirror sticks out. This is due to the `refs` fetch on the mirror repository, `+refs/*:refs/*`, which will fetch all refs from the remote and not just `refs/heads/*` and `refs/tags/*` as a normal repository (and bare) does. The many different ref namespaces on the Jgit repository is because the Jgit repository is managed by Gerrit Code Review that uses different namespaces for some repository specific content, such as `changes` branches for all commits submitted for code review, metadata on code review score, and so on.

The `mirror` repositories are good to know when you would like a quick way to back up a Git repository, and make sure you have everything included without needing more access than the Git access to the machine that hosts the Git repository.

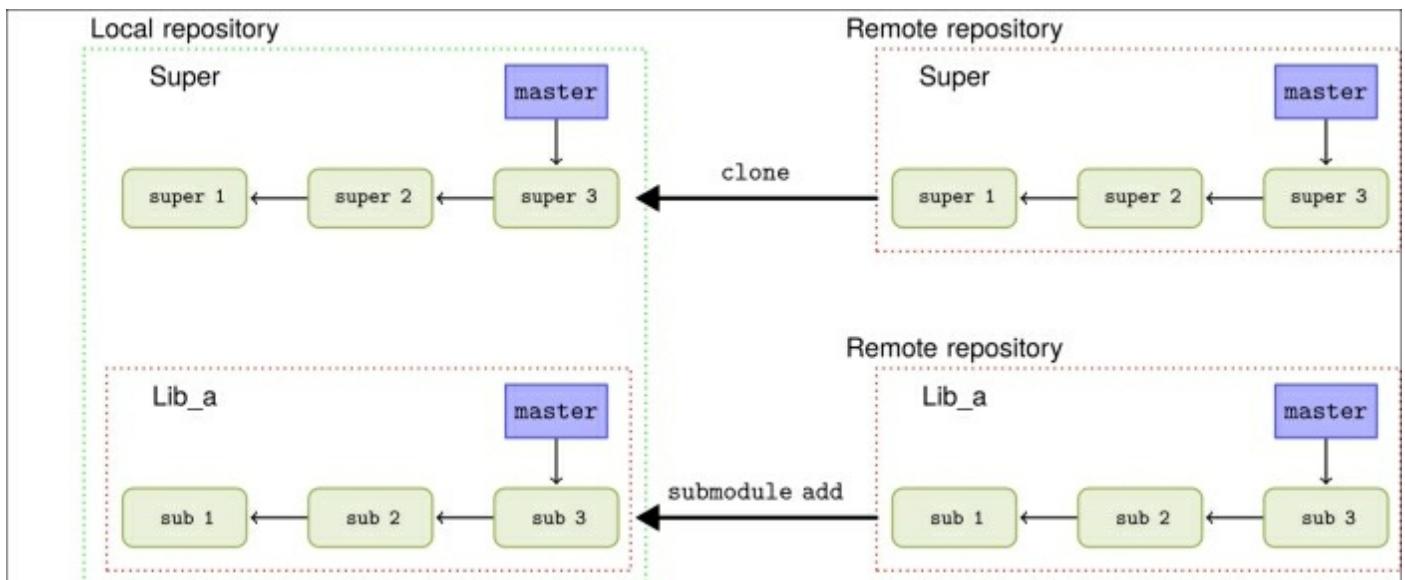
There's more...

The repositories on GitHub store extra information in some `refs` namespaces. If a repository has had a pull request made, the pull request will be recorded in the `refs/pull/*` namespace. Let's look at it in the following code:

```
$ git clone --mirror git@github.com:jenkinsci/extreme-feedback-
plugin.git
$ cd extreme-feedback-plugin.git
$ git show-ref | cut -f2 -d " " | cut -f1,2 -d / | sort -u
refs/heads
refs/meta
refs/pull
refs/tags
```

A quick submodule how-to

When developing a software project, you sometimes find yourself in a situation where you need to use another project as a subpart of your project. This other project can be anything from the another project you are developing to a third-party library. You want to keep the projects separate even though you need to use one project from the other. Git has a mechanism for this kind of project dependency called submodules. The basic idea is that you can clone another Git repository into your project as a subdirectory, but keep the commits from the two repositories separate, as shown in the following diagram:



Getting ready

We'll start by cloning an example repository to be used as the super project:

```
$ git clone https://github.com/dvaske/super.git  
$ cd super
```

How to do it...

1. We'll add a subproject, `lib_a`, to the super project as a Git submodule:

```
$ git submodule add git://github.com/dvaske/lib_a.git lib_a
Cloning into 'lib_a'...
remote: Counting objects: 18, done.
remote: Compressing objects: 100% (14/14), done.
remote: Total 18 (delta 4), reused 17 (delta 3)
Receiving objects: 100% (18/18), done.
Resolving deltas: 100% (4/4), done.
Checking connectivity... done.
```

2. Let's check `git status` using the following command:

```
$ 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)

    new file:   .gitmodules
    new file:   lib_a
```

3. We can take a closer look at the two files in the Git index; `.gitmodules` is a regular file, so we can use `cat`:

```
$ cat .gitmodules
[submodule "lib_a"]
  path = lib_a
  url = git://github.com/dvaske/lib_a.git
$ git diff --cached lib_a
diff --git a/lib_a b/lib_a
new file mode 160000
index 0000000..0d96e7c
--- /dev/null
+++ b/lib_a
@@ -0,0 +1 @@
+Subproject commit 0d96e7cf4d4db64002e63af0f7325d33bdaf84f
```

The `.gitmodules` files contains information about all the submodules registered in the repository. The `lib_a` file stores which commit the submodule's `HEAD` is pointing to when added to the super project. Whenever the submodule is updated with new commits (created locally or fetched), the super project will state the submodule as changed while running `git status`. If the changes to the submodule can be accepted, the submodule revision in the super project is

updated by adding the submodule file and committing this to the super project.

4. We'll update the submodule, `lib_a`, to the latest change on the develop branch using the following command:

```
$ cd lib_a
$ git checkout develop
Branch develop set up to track remote branch develop from
origin by rebasing.
Switched to a new branch 'develop'
$ cd ..
$ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

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:   lib_a (new commits)

no changes added to commit (use "git add" and/or "git
commit -a")
```

5. Let's just check whether there are any updates to the submodule:

```
$ git submodule update
Submodule path 'lib_a': checked out
'0d96e7cf4d4db64002e63af0f7325d33bdaf84f'
```

6. Oops! Now we actually reset our submodule to the state as described in the file for that submodule. We need to switch to the submodule again, check develop, and create a commit in the super project this time:

```
$ cd lib_a
$ git status
HEAD detached at 0d96e7c
nothing to commit, working directory clean
$ git checkout develop
Previous HEAD position was 0d96e7c... Fixes book title in
README
```

```

Switched to branch 'develop'
Your branch is up-to-date with 'origin/develop'.
$ cd ..
$ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

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:   lib_a (new commits)

no changes added to commit (use "git add" and/or "git
commit -a")
$ git add lib_a
$ git commit -m 'Updated lib_a to newest version'
[master 4d371bb] Updated lib_a to newest version
 1 file changed, 1 insertion(+), 1 deletion(-)

```

Notice that the submodule is on default in a detached head state, which means that `HEAD` is pointing directly to a commit instead of a branch. You can still edit the submodule and record commits. However, if you perform a submodule update in the super repository without first committing a new submodule state, your changes can be hard to find. So, always remember to checkout or create a branch while switching to a submodule to work, then you can just checkout the branch again and get your changes back. Since Git Version 1.8.2, it has been possible to make submodules track a branch rather than a single commit. Git 1.8.2 was released on the March 13, 2013, and you can check your version by running `git --version`.

7. To make Git track the branch of a submodule rather than a specific commit, we need to write the name of the branch we want to track in the `.gitmodules` file for the submodule; here we'll use the `stable` branch:

```

$ git config -f .gitmodules submodule.lib_a.branch stable
$ cat .gitmodules
[submodule "lib_a"]

```

```
path = lib_a
url = git://github.com/dvaske/lib_a.git
branch = stable
```

8. We can now add and commit the submodule, and then try to update it using the following command:

```
$ git add .gitmodules
$ git commit -m 'Make lib_a module track its stable branch'
[master bf9b9ba] Make lib_a module track its stable branch
 1 file changed, 1 insertion(+)
$ git submodule update --remote
Submodule path 'lib_a': checked out
'8176a16db21a48a0969e18a51f2c2fb1869418fb'
$ git status
On branch master
Your branch is ahead of 'origin/master' by 3 commits.
  (use "git push" to publish your local commits)

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:   lib_a (new commits)

no changes added to commit (use "git add" and/or "git
commit -a")
```

The submodule is still in the detached HEAD state. However, when updating the submodule with `git submodule update --remote`, changes from the submodule's remote repository will be fetched and the submodule will be updated to the latest commit on the branch it is tracking. We still need to record a commit to the super repository, specifying the state of the submodule.

There's more...

When you are cloning a repository that contains one or more submodules, you need to explicitly fetch them after the clone. We can try this with our newly created submodule repository:

```
$ git clone super super_clone
Cloning into 'super_clone'...
done.
```

Now, initialize and update the submodules:

```
$ cd super_clone
$ git submodule init
Submodule 'lib_a' (git://github.com/dvaske/lib_a.git)
registered for path 'lib_a'
$ git submodule update --remote
Cloning into 'lib_a'...
remote: Counting objects: 18, done.
remote: Compressing objects: 100% (14/14), done.
remote: Total 18 (delta 4), reused 17 (delta 3)
Receiving objects: 100% (18/18), done.
Resolving deltas: 100% (4/4), done.
Checking connectivity... done.
Submodule path 'lib_a': checked out
'8176a16db21a48a0969e18a51f2c2fb1869418fb'
```

The repository is ready for development!

Tip

When cloning the repository, the submodules can be initialized and updated directly after the clone if the `--recursive` or `--recurse-submodules` option is given.

Subtree merging

An alternative to submodules is subtree merging. Subtree merging is a strategy that can be used while performing merges with Git. The subtree merge strategy is useful when merging a branch (or as we'll see in this recipe another project) into a subdirectory of a Git repository instead of the root directory. When using the subtree merge strategy, the history of the subproject is joined with the history of the super project, while the subproject's history can be kept clean except for commits indented to go upstream.

Getting ready

We'll use the same repositories as in the last recipe, and we'll reclone the super project to get rid of the submodule setup:

```
$ git clone https://github.com/dvaske/super.git  
$ cd super
```

How to do it...

We'll add the subproject as a new remote and fetch the history:

```
$ git remote add lib_a git://github.com/dvaske/lib_a.git  
$ git fetch lib_a  
warning: no common commits  
remote: Reusing existing pack: 18, done.  
remote: Total 18 (delta 0), reused 0 (delta 0)  
Unpacking objects: 100% (18/18), done.  
From git://github.com/dvaske/lib_a  
 * [new branch]      develop    -> lib_a/develop  
 * [new branch]      master     -> lib_a/master  
 * [new branch]      stable     -> lib_a/stable
```

We can now create a local branch, `lib_a_master`, which points to the same commit as the `master` branch in `lib_a` (`lib_a/master`):

```
$ git checkout -b lib_a_master lib_a/master  
Branch lib_a_master set up to track remote branch master from  
lib_a by rebasing.
```

```
Switched to a new branch 'lib_a_master'
```

We can check the content of our working tree using the following command:

```
$ ls  
README.md  a.txt
```

If we switch back to the `master` branch, we should see the content of the super repository in our directory:

```
$ git checkout master  
Switched to branch 'master'  
Your branch is up-to-date with 'origin/master'.  
$ ls  
README.md  super.txt
```

Git changes branches and populates the working directory as normal even though the branches are originally from two different repositories. Now, we want to merge the history from `lib_a` into a subdirectory. First, we prepare a merge commit by merging with the `ours` strategy and make sure the commit isn't completed (we need to bring in all the files).

```
$ git merge -s ours --no-commit lib_a_master  
Automatic merge went well; stopped before committing as  
requested
```

In short what the `ours` strategy tells Git to do the following: Merge in this branch, but keep the resulting tree the same as the tree on the tip of this branch. So, the branch is merged, but all the changes it introduced are discarded. In our previous command line, we also passed the `--no-commit` option. This option stops Git from completing the merge, but leaves the repository in a merging state. We can now add the content of the `lib_a` repository to the `lib_a` folder in the repository root. We do this with `git read-tree` to make sure the two trees are exactly the same as follows:

```
$ git read-tree --prefix=lib_a/ -u lib_a_master
```

Our current directory structure looks as follows:

```
$ tree
.
|-- README.md
|-- lib_a
|   |-- README.md
|   '-- a.txt
'-- super.txt
```

It is time to conclude the merge commit we started using the following command:

```
$ git commit -m 'Initial add of lib_a project'
[master 5066b7b] Initial add of lib_a project
```

Now the subproject is added. Next, we'll see how we can update the super project with new commits from the subproject and how to copy commits made in the super project to the subproject.

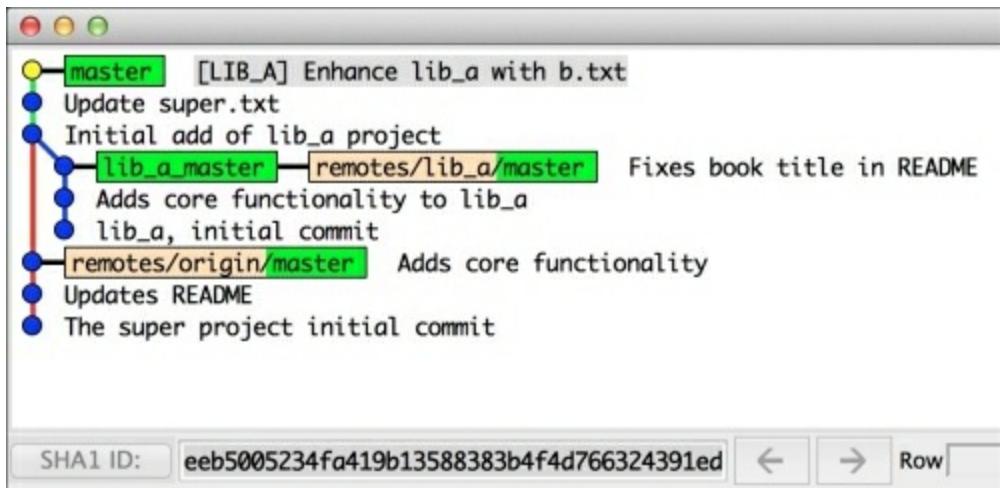
We need to add and commit some changes to the super project using the following command:

```
$ echo "Lib_a included\!" >> super.txt
$ git add super.txt
$ git commit -m "Update super.txt"
[master 83ef9a4] Update super.txt
 1 file changed, 1 insertion(+)
```

Some changes are made to the subproject committed in the super project:

```
$ echo "The b file in lib_a" >> lib_a/b.txt
$ git add lib_a/b.txt
$ git commit -m "[LIB_A] Enhance lib_a with b.txt"
[master debe836] [LIB_A] Enhance lib_a with b.txt
 1 file changed, 1 insertion(+)
 create mode 100644 lib_a/b.txt
```

The current history looks like the following screenshot:

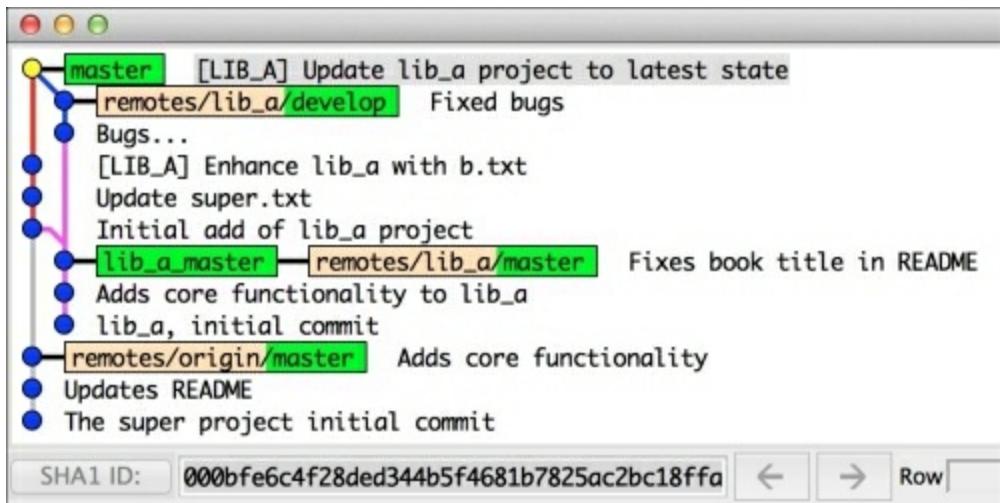


The merge can be seen in the previous screenshot and the two root commits of the repository, the original root commit and the root from `lib_a`.

Now, we will learn to integrate new commits into the super repository made in the subproject, `lib_a`. Normally, we would do this by checking out the `lib_a_master` branch and performing `pull` on this to get the latest commit from the remote repository. However, as we are working with example repositories in this recipe, no new commits are available on the `master` branch. Instead, we'll use the `develop` and `stable` branches from `lib_a`. We'll now integrate commits from the `develop` branch on `lib_a`. We do this directly using the `lib_a/develop` reference in the repository as follows:

```
$ git merge -m '[LIB_A] Update lib_a project to latest state' -s \ subtree lib_a/develop
Merge made by the 'subtree' strategy.
 lib_a/a.txt | 2 ++
 1 file changed, 2 insertions(+)
```

Our `master` branch has now been updated with the commits from `lib_a/develop` as shown in the following screenshot:



Now, it is time to add the commits we made in the `lib_a` directory back to the `lib_a` project. First, we'll change the `lib_a_master` branch and merge that with `lib_a/develop` to be as up to date as possible:

```
$ git checkout lib_a_master
$ git merge lib_a/develop
Updating 0d96e7c..ab47aca
Fast-forward
 a.txt | 2 ++
 1 file changed, 2 insertions(+)
```

Now we are ready to merge changes from the super project to the subproject. In order not to merge the history of the super project to the subproject, we'll use the `--squash` option. This option stops Git from completing the merge, and unlike the previous case where we also stopped a merge from recording a commit, it does not leave the repository in a merging state. The state of the working directory and staging area are, however, set as though a real merge has happened.

```
$ git merge --squash -s subtree --no-commit master
Squash commit -- not updating HEAD
Automatic merge went well; stopped before committing as
requested
```

Now, we can record a commit with all the changes done in `lib_a` from the super project:

```
$ git commit -m 'Enhance lib_a with b.txt'
```

```
[lib_a_master 01e45f7] Enhance lib_a with b.txt
1 file changed, 1 insertion(+)
create mode 100644 b.txt
```

The history for the `lib_a` repository is seen in the following screenshot:

```
lib_a_master Enhance lib_a with b.txt
remotes/lib_a/develop Fixed bugs
Bugs...
remotes/lib_a/master Fixes book title in README
Adds core functionality to lib_a
lib_a, initial commit

SHA1 ID: 01e45f778fb79ba635a0b0edf1463a9a10e5a8db
```

We can integrate more changes from `lib_a/stable` into the super project, but first we'll update the `lib_a_master` branch so we can integrate them from here:

```
$ git merge lib_a/stable
Merge made by the 'recursive' strategy.
 a.txt | 2 ++
 1 file changed, 2 insertions(+)
```

A new commit was added to the subproject, as shown in the following screenshot:

```
lib_a_master Merge remote-tracking branch 'lib_a/stable' into lib_a_master
remotes/lib_a/stable Added stability to lib
Enhance lib_a with b.txt
remotes/lib_a/develop Fixed bugs
Bugs...
remotes/lib_a/master Fixes book title in README
Adds core functionality to lib_a
lib_a, initial commit

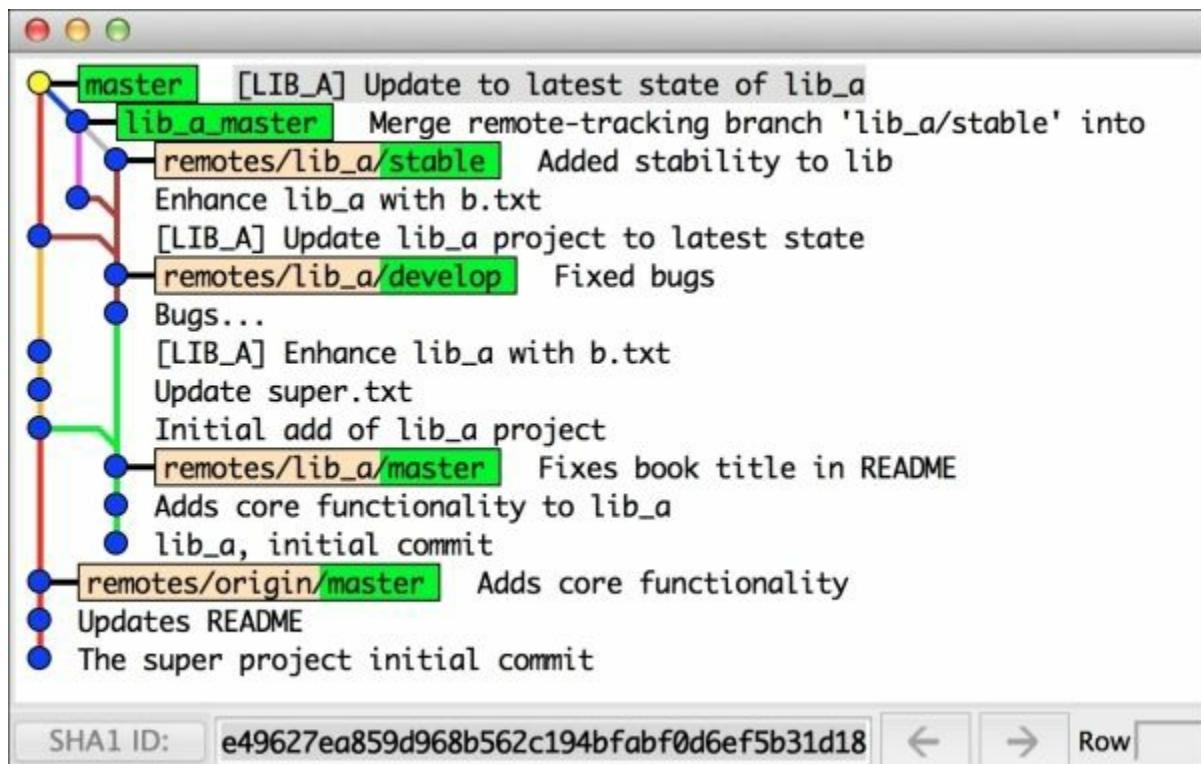
SHA1 ID: 9d3195cf5e5e9d722d535168635ce2344e4bfae1
```

The last task is to integrate the new commit on `lib_a_master` to the

master branch in the super repository. This is done as in the previous case with the `subtree` strategy option to `git merge`:

```
$ git checkout master
$ git merge -s subtree -m '[LIB_A] Update to latest state of
lib_a' lib_a_master
Merge made by the 'subtree' strategy.
 lib_a/a.txt | 2 ++
 1 file changed, 2 insertions(+)
```

The resulting history is shown in the following screenshot:



How it works...

When using the subtree strategy, Git finds out which subtree in your repository, the branch you are trying to merge fits in. This is why we added the content of the `lib_a` repository with the `read-tree` command to make sure we got the exact same SHA-1 ID for the `lib_a` directory in the super project as the root tree in the `lib_a` project.

We can verify this by finding the SHA-1 of the `lib_a` tree in the super project in the commit where we merged in the subproject:

```
$ git ls-tree a3662eb94abf0105a25309653b5d2ce67a4028d2
100644 blob 456a5df638694a699fff7a7ff31a496630b12d01 README.md
040000 tree 7d66ad11cb22c6d101c7ac9c309f7dce25231394 lib_a
100644 blob c552dead26fdbba634c91d35708f1cfcc2c4b2a100 super.txt
```

The ID of the root tree at `lib_a/master` can be found out by using the following command:

```
$ git cat-file -p lib_a/master
tree 7d66ad11cb22c6d101c7ac9c309f7dce25231394
parent a7d76d9114941b9d35dd58e42f33ed7e32a9c134
author Aske Olsson <aske.olsson@switch-gears.dk> 1396553189
+0200
committer Aske Olsson <aske.olsson@switch-gears.dk> 1396553189
+0200
```

Fixes book title in README

See also

Another way of using subtree merging is with the `git subtree` command. This is not enabled by default in Git installations, but is distributed with Git since 1.7.11. You can see how to install and use it at the following links:

- For installation, go to
<https://github.com/git/git/blob/master/contrib/subtree/INSTALL>
- To understand how to use a subtree, go to
<https://github.com/git/git/blob/master/contrib/subtree/git-subtree.txt>

Submodule versus subtree merging

There is no easy answer to the question of whether or not to use submodules or subtree merging for a project. When choosing submodule, a lot of extra pressure is put on the developers working on the project as they need to make sure they keep the submodule and the super project in sync. When choosing to add a project by subtree merging, no or little extra complexity is added for developers. The repository maintainer, however, needs to make sure the subproject is up to date and that commits are added back to the subproject. Both methods work and are in use, and it is probably just a matter of getting used to either method before finding it natural to use. A completely different solution is to use the build system of the super project to fetch the necessary dependencies, as for example, **Maven** or **Gradle** does.

Chapter 10. Patching and Offline Sharing

In this chapter, we will cover the following topics:

- Creating patches
- Creating patches from branches
- Applying patches
- Sending patches
- Creating Git bundles
- Using a Git bundle
- Creating archives from a tree

Introduction

With the distributed nature of Git and the many existing hosting options available for Git, it is very easy to share the history between machines when they are connected through a network. In cases where the machines that need to share history are not connected or can't use the transport mechanisms supported, Git provides other methods to share the history. Git provides an easy way to format patches from the existing history, sending those with an e-mail and applying them to another repository. Git also has a bundle concept, where a bundle that contains only part of the history of a repository can be used as a remote for another repository. Finally, Git provides a simple and easy way to create an archive for a snapshot of the folder/subfolder structure for a given reference.

With these different methods provided by Git, it becomes easy to share the history between repositories, especially where the normal push/pull methods are not available.

Creating patches

In this recipe, we'll learn how to make patches out of commits. Patches can be sent via e-mails for quick sharing or copied to sneakernet devices (USB sticks, memory cards, external hard disk drives, and so on) if they need to be applied on an offline computer or the like. Patches can be useful methods to review code as the reviewer can apply the patch on his repository, investigate the diff, and check the program. If the reviewer decides the patch is good, he can publish (`push`) it to a public repository, given the reviewer is the maintainer of the repository. If the reviewer rejects the patch, he can simply reset his branch to the original state and inform the author of the patch that more work is needed before the patch can be accepted.

Getting ready

In this example, we'll clone and use a new repository. The repository is just an example repository for the Git commands and only contains some example commits:

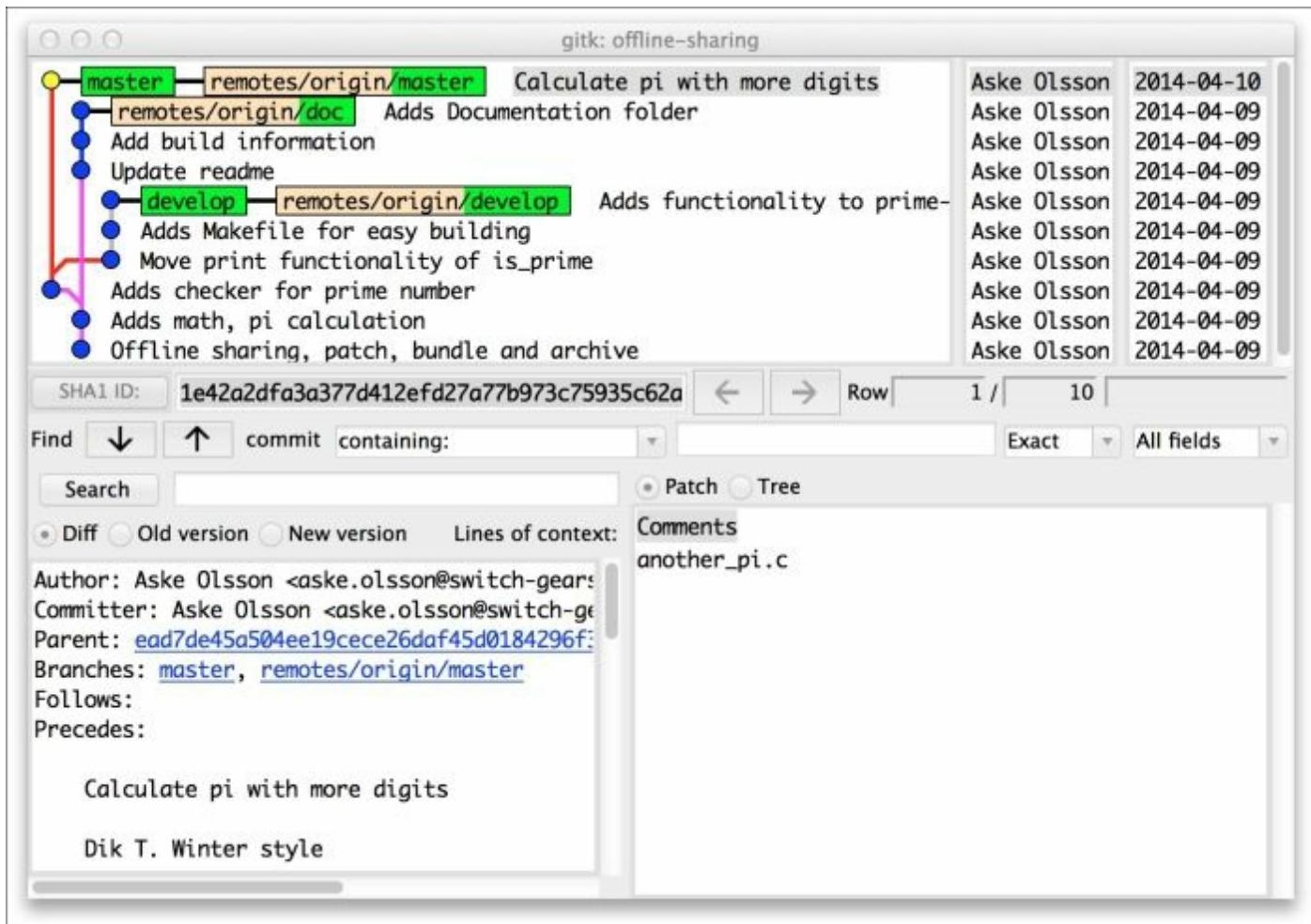
```
$ git clone https://github.com/dvaske/offline-sharing.git  
$ cd offline-sharing  
$ git checkout master
```

How to do it...

Let's see the history of the repository as shown by Gitk:

```
$ gitk --all
```

The history of the repository can be seen in the following screenshot:



There are three branches in the repository: master, develop, and doc. All of them differ by one or more commits to the others. On the master branch, we can now create a patch file for the latest commit on the master branch and store it in the latest-commit folder, as shown in the following command:

```
$ git format-patch -1 -o latest-commit
latest-commit/0001-Calculate-pi-with-more-digits.patch
```

If we look at the file created by the patch command, we will see the following command:

```
$ cat latest-commit/0001-Calculate-pi-with-more-digits.patch
```

```
From 1e42a2dfa3a377d412efd27a77b973c75935c62a Mon Sep 17
00:00:00 2001
From: Aske Olsson <aske.olsson@switch-gears.dk>
Date: Thu, 10 Apr 2014 09:19:29 +0200
```

Subject: [PATCH] Calculate pi with more digits

Dik T. Winter style

Build: gcc -Wall another_pi.c -o pi

Run: ./pi

```
another_pi.c | 21 ++++++-----+
1 file changed, 21 insertions(+)
create mode 100644 another_pi.c
```

diff --git a/another_pi.c b/another_pi.c

new file mode 100644

index 000000..86df41b

--- /dev/null

+++ b/another_pi.c

@@ -0,0 +1,21 @@

+/* Pi with 800 digits

+ * Dik T. Winter style, but modified slightly

+ * <https://crypto.stanford.edu/pbc/notes/pi/code.html>

+ */

+ #include <stdio.h>

+

+void another_pi (void) {

+ printf("800 digits of pi:\n");

+ int a=10000, b=0, c=2800, d=0, e=0, f[2801], g=0;

+ for (;b<c;)f[b++]=a/5;

+ for (;d=0,g=c*2;c-=14,printf("%.4d",e+d/a),e=d%a)

+ for (b=c; d+=f[b]*a, f[b]=d%--g,d/=g--,--b; d*=b);

+

+ printf("\n");

}

+

+int main (void) {

+ another_pi();

+

+ return 0;

}

--

1.9.1

The previous snippet is the content of the patch file produced. It contains a header much like an e-mail with the From, Date, and Subject fields, a body with the commit message and after the 3 dashes (---), the actual patch, and finally ending with 2 dashes (--), and the Git version

used to generate the patch. The patch generated by `git format-patch` is in the **UNIX** mailbox format but with a magic fixed timestamp to identify that it comes from `git format-patch` rather than a real mailbox. You can see the time stamp in the first line after the `sha-1 ID Mon Sep 17 00:00:00 2001`.

How it works...

When generating the patch, Git will diff the commit at HEAD with its parent commit and use this diff as the patch. The `-1` option tells Git only to generate patches for the last commit and `-o latest-commit` tells Git to store the patch in the folder `latest-commit`. The folder will be created if it does not exist.

There's more...

If you want to create patches for several commits, say the last 3 commits, you just pass on `-3` to `git format-patch` instead of the `-1`.

Format the latest three commits as patches in the `latest-commits` folder:

```
$ git format-patch -3 -o latest-commits
latest-commits/0001-Adds-math-pi-calculation.patch
latest-commits/0002-Adds-checker-for-prime-number.patch
latest-commits/0003-Calculate-pi-with-more-digits.patch
$ ls -la latest-commits
total 24
drwxr-xr-x  5 aske  staff   170 Apr 11 21:55 .
drwxr-xr-x  8 aske  staff   272 Apr 11 21:55 ..
-rw-r--r--  1 aske  staff   676 Apr 11 21:55 0001-Adds-math-pi-
calculation.patch
-rw-r--r--  1 aske  staff  1062 Apr 11 21:55 0002-Adds-checker-
for-prime-number.patch
-rw-r--r--  1 aske  staff  1041 Apr 11 21:55 0003-Calculate-pi-
with-more-digits.patch
```

Creating patches from branches

Instead of counting the number of commits you need to make patches for, you can create the patches by specifying the target branch when running the format-patch command.

Getting ready

We'll use the same repository as in the previous example:

```
$ git clone https://github.com/dvaske/offline-sharing.git  
$ cd offline-sharing
```

Create the master branch locally without checking out:

```
$ git branch master origin/master
```

Make sure we have develop checked out:

```
$ git checkout develop
```

How to do it...

We'll pretend that we have been working on the `develop` branch and have made some commits. Now, we need to format patches for all these commits so we can send them to the repository maintainer or carry them to another machine.

Let's see the commits on `develop` not on `master`:

```
$ git log --oneline master..develop  
c131c8b Adds functionality to prime-test a range of numbers  
274a7a8 Adds Makefile for easy building  
88798da Move print functionality of is_prime
```

Now, instead of running `git format-patch -3` to get patches made for these 3 commits, we'll tell Git to create patches for all the commits that are not on the `master` branch:

```
$ git format-patch -o not-on-master master
```

```
not-on-master/0001-Move-print-functionality-of-is_prime.patch  
not-on-master/0002-Adds-Makefile-for-easy-building.patch  
not-on-master/0003-Adds-functionality-to-prime-test-a-range-of-numbers.patch
```

How it works...

Git makes a list of commits from `develop` that are not on the `master` branch, much like we did before creating the patches, and makes patches for these. We can check the content of the folder `not-on-master`, which we specified as the output folder (`-o`) and verify that it contains the patches as expected:

```
$ ls -1 not-on-master  
0001-Move-print-functionality-of-is_prime.patch  
0002-Adds-Makefile-for-easy-building.patch  
0003-Adds-functionality-to-prime-test-a-range-of-numbers.patch
```

There's more...

The `git format-patch` command has many options and besides the `-<n>` option to specify the number of commits in order to create patches for and the `-o <dir>` for the target directory, some useful options are as follows:

- `-s, --signoff`: Adds a `Signed-off-by` line to the commit message in the patch file with the name of the committer. This is often required when mailing patches to the repository maintainers. This line is required for patches to be accepted when they are sent to the Linux kernel mailing list and the Git mailing list.
- `-n, --numbered`: Numbers the patch in the subject line as `[PATCH n/m]`.
- `--suffix=.<sfx>`: Sets the suffix of the patch; it can be empty and does not have to start with a dot.
- `-q, --quiet`: Suppresses the printing of patch filenames when generating patches.
- `--stdout`: Prints all commits to the standard output instead of creating files.

Applying patches

Now we know how to create patches from commits. It is time to learn to apply them.

Getting ready

We'll use the repository from the previous examples along with the generated patches as follows:

```
$ cd offline-sharing
$ git checkout master
$ ls -la
.
..
.git
Makefile
README.md
another_pi.c
latest-commit
math.c
not-on-master
```

How to do it...

First, we'll checkout the `develop` branch and apply the patch generated from the `master` branch (`0001-Calculate-pi-with-more-digits.patch`) in the first example. We use the Git command `am` to apply the patches; `am` is short for `apply from mailbox`:

```
$ git checkout develop
Your branch is up-to-date with 'origin/develop'.
$ git am latest-commit/0001-Calculate-pi-with-more-digits.patch
Applying: Calculate pi with more digits
```

We can also apply the `master` branch to the series of patches that was generated from the `develop` branch as follows:

```
$ git checkout master
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.
```

```
$ git am not-on-master/*
Applying: Move print functionality of is_prime
Applying: Adds Makefile for easy building
Applying: Adds functionality to prime-test a range of numbers
```

How it works...

The `git am` command takes the mailbox file specified in the input and applies the patch in the file to the files needed. Then, a commit is recorded using the commit message and author information from the patch. The committer identity of the commit will be the identity of the person performing the `git am` command. We can see the author and committer information with `git log`, but we need to pass the `--pretty=fuller` option to also view the committer information:

```
$ git log -1 --pretty=fuller
commit 5af5ee2746b67893b0550d8a63110c48fd1b667c
Author: Aske Olsson <aske.olsson@switch-gears.dk>
AuthorDate: Wed Apr 9 21:50:18 2014 +0200
Commit: Aske Olsson <aske.olsson@switch-gears.dk>
CommitDate: Fri Jun 13 22:50:45 2014 +0200
```

Adds functionality to prime-test a range of numbers

There's more...

The `git am` command applies the patches in the files specified and records the commits in the repository. However, if you only want to apply the patch to the working tree or the staging area and not record a commit, you can use the `git apply` command.

We can try to apply the patch from the `master` branch to the `develop` branch once again; we just need to reset the `develop` branch first:

```
$ git checkout develop
Switched to branch 'develop'
Your branch is ahead of 'origin/develop' by 1 commit.
  (use "git push" to publish your local commits)
$ git reset --hard origin/develop
HEAD is now at c131c8b Adds functionality to prime-test a range
of numbers
$ git apply latest-commit/0001-Calculate-pi-with-more-
```

`digits.patch`

Now, we can check the state of the repository with the `status` command:

```
$ git status
On branch develop
Your branch is up-to-date with 'origin/develop'.

Untracked files:
  (use "git add <file>..." to include in what will be
committed)

another_pi.c
latest-commit/
not-on-master/

nothing added to commit but untracked files present (use "git
add" to track)
```

We successfully applied the patch to the working tree. We can also apply it to the staging area and the working tree using the `--index` option, or only to the staging area using the `--cached` option.

Sending patches

In the previous example, you saw how to create and apply patches. You can, of course, attach these patch files directly to an e-mail, but Git provides a way to send the patches directly as e-mails with the `git send-email` command. The command requires some setting up, but how you do that is heavily dependent on your general mail and SMTP configuration. A general guide can be found in the Git help pages or visit <http://git-scm.com/docs/git-send-email>.

Getting ready

We'll set up the same repository as in the previous example:

```
$ git clone https://github.com/dvaske/offline-sharing.git
Cloning into 'offline-sharing'...
remote: Counting objects: 32, done.
remote: Compressing objects: 100% (25/25), done.
remote: Total 32 (delta 7), reused 30 (delta 6)
Unpacking objects: 100% (32/32), done.
Checking connectivity... done.
$ cd offline-sharing
```

How to do it...

First, we'll send the same patch as the one we created in the first example. We'll send it to ourselves using the e-mail address we specified in our Git configuration. Let's create the patch again with `git format-patch` and send it with `git send-email`:

```
$ git format-patch -1 -o latest-commit
latest-commit/0001-Calculate-pi-with-more-digits.patch
```

Save the e-mail address from the Git configuration to a variable as follows:

```
$ emailaddr=$(git config user.email)
```

Send the patch using the e-mail address in both the to and from fields:

```
$ git send-email --to $emailaddr --from $emailaddr latest-
commit/0001-Calculate-pi-with-more-digits.patch
latest-commit/0001-Calculate-pi-with-more-digits.patch
(mbox) Adding cc: Aske Olsson <aske.olsson@switch-gears.dk>
from line 'From: Aske Olsson <aske.olsson@switch-gears.dk>'
OK. Log says:
```

```
Server: smtp.gmail.com
MAIL FROM:<aske.olsson@switch-gears.dk>
RCPT TO:<aske.olsson@switch-gears.dk>
From: aske.olsson@switch-gears.dk
To: aske.olsson@switch-gears.dk
Subject: [PATCH] Calculate pi with more digits
Date: Mon, 14 Apr 2014 09:00:11 +0200
Message-ID: <1397458811-13755-1-git-send-email-
aske.olsson@switch-gears.dk>
X-Mailer: git-send-email 1.9.1
```

An e-mail check will reveal an e-mail in the inbox, as shown in the following screenshot:



More ▾

[PATCH] Calculate pi with more digits



Inbox x



Aske Olsson <aske.olsson@switch-gear

10:54 PM (10 hours ago)



to aske ▾

From: Aske Olsson <aske.olsson@switch-gears.dk>

Dik T. Winter style

Build: gcc -Wall another_pi.c -o pi

Run: ./pi

another_pi.c | 21 ++++++-----

1 file changed, 21 insertions(+)

create mode 100644 another_pi.c

diff --git a/another_pi.c b/another_pi.c

new file mode 100644

index 0000000..86df41b

--- /dev/null

+++ b/another_pi.c

@@@ -0,0 +1,21 @@@

+/* Pi with 800 digits

+ * Dik T. Winter style, but modified slightly

+ * <https://crypto.stanford.edu/pbc/notes/pi/code.html>

+ */

+ #include <stdio.h>

+

+void another_pi (void){

+ printf("800 digits of pi:\n");

+ int a=10000, b=0, c=2800, d=0, e=0, f[2801], g=0;

+ for (;b<c;)f[b++]=a/5;

+ for (;d=0,g=c*2;c-=14,printf("%.4d",e+d/a),e=d%a)

+ for (b=c; d+=f[b]*a, f[b]=d%--g,d/=g--,--b; d*=b);

+

+ printf("\n");

+}

+

+int main (void){

+ another_pi();

+

+ return 0;

+}

--

1.9.1

How it works...

As we saw in the previous examples, `git format-patch` creates the patch files in the Unix mbox format, so only a little extra effort is required to allow Git to send the patch as an e-mail. When sending e-mails with `git send-email`, make sure your **MUA (Mail User Agent)** does not break the lines in the patch files, replace tabs with spaces, and so on. You can test this easily by sending a patch to yourself and checking whether it can be applied cleanly to your repository.

There's more...

The `send-email` command can of course be used to send more than one patch at a time. If a directory is specified instead of a single patch file, all the patches in that directory will be sent. We don't even have to generate the patch files before sending them; we can just specify the same range of revisions we want to send as we would have specified for the `format-patch` command. Then, Git will create the patches on the fly and send them. When we send a series of patches this way, it is good practice to create a cover letter with a bit of explanation about the patch series that follows. The cover letter can be created by passing `--cover-letter` to the `send-email` command. We'll try sending patches for the commits on `develop` since it is branched from `master` (the same patches as in the second example) as follows:

```
$ git checkout develop
Switched to branch 'develop'
Your branch is up-to-date with 'origin/develop'.
$ git send-email --to aske.olsson@switch-gears.dk --from \
aske.olsson@switch-gears.dk --cover-letter --annotate
origin/master
/tmp/path/for/patches/0000-cover-letter.patch
/tmp/path/for/patches/0001-Move-print-functionality-of-
is_prime.patch
/tmp/path/for/patches/0002-Adds-Makefile-for-easy-
building.patch
/tmp/path/for/patches/0003-Adds-functionality-to-prime-test-a-
range-of-numbers.patch
(mbox) Adding cc: Aske Olsson <aske.olsson@switch-gears.dk>
from line 'From: Aske Olsson <aske.olsson@switch-gears.dk>'
```

OK. Log says:

```
Server: smtp.gmail.com
MAIL FROM:<aske.olsson@switch-gears.dk>
RCPT TO:<aske.olsson@switch-gears.dk>
From: aske.olsson@switch-gears.dk
To: aske.olsson@switch-gears.dk
Subject: [PATCH 0/3] Cover Letter describing the patch series
Date: Sat, 14 Jun 2014 23:35:14 +0200
Message-ID: <1397459884-13953-1-git-send-email-
aske.olsson@switch-gears.dk>
X-Mailer: git-send-email 1.9.1
...
...
```

We can check our e-mail inbox and see the four mails we sent: the cover letter and the 3 patches, as shown in the following screenshot:

[PATCH 3/3] Adds functionality to prime-test a range of numbers - From: Aske Olsson <aske.olsson@switch-gears.dk> --- math	11:35 pm
[PATCH 2/3] Adds Makefile for easy building - From: Aske Olsson <aske.olsson@switch-gears.dk> --- Makefile 10 ++++++++	11:35 pm
[PATCH 1/3] Move print functionality of is_prime - From: Aske Olsson <aske.olsson@switch-gears.dk> Moves print functionality !	11:35 pm
[PATCH 0/3] Cover Letter describing the patch series - From: Aske Olsson <aske.olsson@switch-gears.dk> Best math I ever did	11:35 pm
[PATCH] Adds functionality to prime-test a range of numbers - From: Aske Olsson <aske.olsson@switch-gears.dk> --- math.c	11:32 pm

Before sending the patches, the cover letter is filled out and by default has [PATCH 0/3] (if sending 3 patches) in the subject line. A cover letter with only the default template subject and body won't be sent as default. In the scripts that come with this chapter, the `git send-email` command invokes the `--force` and `--confirm=never` options. This was done for script automation to force Git to send the mails even though the cover letter has not been changed from the default. You can try to remove these options, put in the `--annotate` option, and run the scripts again. You should then be able to edit the cover letter and e-mails that contain the patches before sending.

Creating Git bundles

Another method to share the repository history between repositories is to use the `git bundle` command. A Git bundle is a series of commits that can work as a remote repository, but without having the full history of a repository included in the bundle.

Getting ready

We'll use a fresh clone of the `offline-sharing` repository as follows:

```
$ git clone https://github.com/dvaske/offline-sharing.git  
$ cd offline-sharing  
$ git checkout master
```

How to do it...

First, we'll create a root bundle, as shown in the following command, so that the history in the bundle forms a complete history and the initial commit is also included:

```
$ git bundle create myrepo.bundle master  
Counting objects: 12, done.  
Delta compression using up to 8 threads.  
Compressing objects: 100% (11/11), done.  
Writing objects: 100% (12/12), 1.88 KiB | 0 bytes/s, done.  
Total 12 (delta 1), reused 0 (delta 0)
```

We can verify the bundle content with `git bundle verify`:

```
$ git bundle verify myrepo.bundle  
The bundle contains this ref:  
1e42a2dfa3a377d412efd27a77b973c75935c62a refs/heads/master  
The bundle records a complete history.  
myrepo.bundle is okay
```

To make it easy to remember which commit we included as the latest commit in the bundle, we create a tag that points to this commit; the commit is also pointed to by the `master` branch:

```
$ git tag bundleForOtherRepo master
```

We have created the root bundle that contains the initial commits of the repository history. We can now create a second bundle that contains the history from the tag we just created to the tip of the `develop` branch. Note that in the following command, we use the same name for the bundle file, `myrepo.bundle`, and this will overwrite the old bundle file:

```
$ git bundle create myrepo.bundle bundleForOtherRepo..develop
Counting objects: 12, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (9/9), done.
Writing objects: 100% (9/9), 1.47 KiB | 0 bytes/s, done.
Total 9 (delta 2), reused 0 (delta 0)
```

Note

It might seem strange to overwrite the bundle file just after creating it, but there is some sense in naming the bundle files by the same name. As you will also see in the next recipe, when using a bundle file, you add it to your repository as a remote, the URL being the file path of the bundle. The first time you do this is with the root bundle file and the URL. The file path of the bundle file will be stored as the URL of the remote repository. So, the next time you need to update the repository, you just overwrite the bundle file and perform `fetch` from the repository.

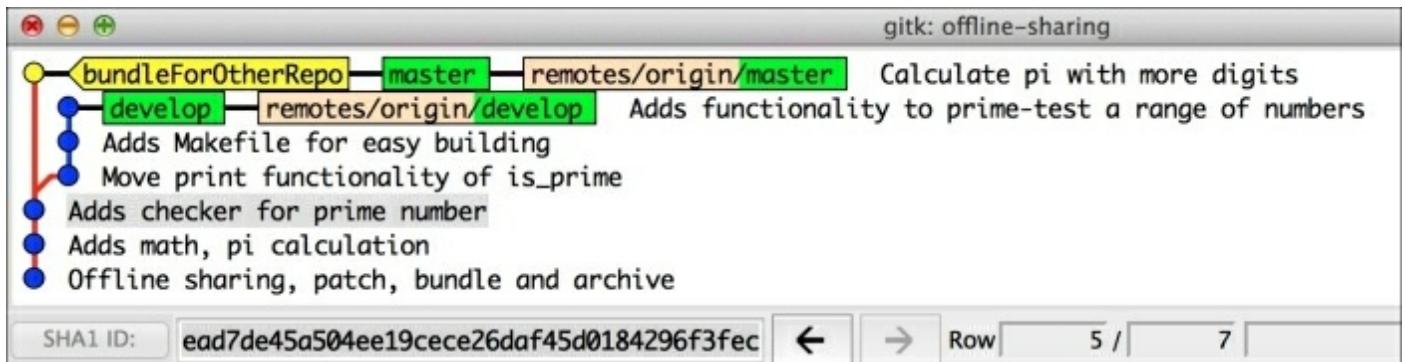
If we verify the bundle, we can see which commit needs to exist in the target repository before the bundle can be used:

```
$ git bundle verify myrepo.bundle
The bundle contains this ref:
c131c8bb2bf8254e46c013bfb33f4a61f9d4b40e refs/heads/develop
The bundle requires this ref:
ead7de45a504ee19cece26daf45d0184296f3fec
myrepo.bundle is okay
```

We can check the history and see that the `ead7de4` commit is where `develop` is branched off so it makes sense that this commit is the basis for the bundle we have just created:

```
$ gitk master develop
```

The previous command gives the following output:



How it works...

The `bundle` command creates a binary file with the history of the specified commit range included. When creating the bundle as a range of commits that does not include the initial commit in the repository (for example, `bundleForOtherRepo..develop`), it is important to make sure the range matches the history in the repository where the bundle is going to be used.

Using a Git bundle

In the last example, we saw how we could create bundles from the existing history that contains a specified range of history. Now, we'll learn to use these bundles either to create a new repository or to add the history to an existing one.

Getting ready

We'll use the same repository and methods as in the last example to create bundles, but we'll recreate them in this example to be able to use them one at a time. First, we'll prepare the repository and the first bundle, as shown in the following commands:

```
$ rm -rf offline-sharing
$ git clone https://github.com/dvaske/offline-sharing.git
Cloning into 'offline-sharing'...
remote: Counting objects: 32, done.
remote: Compressing objects: 100% (25/25), done.
remote: Total 32 (delta 7), reused 30 (delta 6)
Unpacking objects: 100% (32/32), done.
Checking connectivity... done.
$ cd offline-sharing
$ git checkout master
Branch master set up to track remote branch master from origin
by rebasing.
Switched to a new branch 'master'
$ git bundle create myrepo.bundle master
Counting objects: 12, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (11/11), done.
Writing objects: 100% (12/12), 1.88 KiB | 0 bytes/s, done.
Total 12 (delta 1), reused 0 (delta 0)
$ git tag bundleForOtherRepo master
```

How to do it...

Now, let's create a new repository from the bundle file we just created. We can do that with the `git clone` command and by specifying the URL to the remote repository as the path to the bundle. We'll see how to do

that in the following code snippet:

```
$ cd ..
$ git clone -b master offline-sharing/myrepo.bundle offline-other
Cloning into 'offline-other'...
Receiving objects: 100% (12/12), done.
Resolving deltas: 100% (1/1), done.
Checking connectivity... done.
```

The new repository is created in the `offline-other` folder. Let's check the history of that repository by using the following command:

```
$ cd offline-other
$ git log --oneline --decorate --all
1e42a2d (HEAD, origin/master, master) Calculate pi with more digits
ead7de4 Adds checker for prime number
337bfd0 Adds math, pi calculation
7229805 Offline sharing, patch, bundle and archive
```

The repository contains, as expected, all the history of the `master` branch in the original repository. We can now create a second bundle, the same as in the previous example, that contains history from the tag we created (`bundleForOtherRepo`) to the tip of the `develop` branch:

```
$ cd ..
$ cd offline-sharing
$ git bundle create myrepo.bundle bundleForOtherRepo..develop
Counting objects: 12, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (9/9), done.
Writing objects: 100% (9/9), 1.47 KiB | 0 bytes/s, done.
Total 9 (delta 2), reused 0 (delta 0)
$ git bundle verify myrepo.bundle
The bundle contains this ref:
c131c8bb2bf8254e46c013bfb33f4a61f9d4b40e refs/heads/develop
The bundle requires this ref:
ead7de45a504ee19cece26daf45d0184296f3fec
myrepo.bundle is okay
```

As we also saw in the previous example, the bundle requires that the `ead7de45a504ee19cece26daf45d0184296f3fec` commit exists in the repository we'll use with the bundle. Let's check the repository we

created from the first bundle for this commit by using the following command:

```
$ cd ..
$ cd offline-other
$ git show -s ead7de45a504ee19cece26daf45d0184296f3fec
commit ead7de45a504ee19cece26daf45d0184296f3fec
Author: Aske Olsson <aske.olsson@switch-gears.dk>
Date:   Wed Apr 9 21:28:51 2014 +0200
```

Adds checker for prime number

The commit exists. Now we can use the new bundle file as it has the same filename and path as the first bundle we created. We can just use `git fetch` in the `offline-other` repository as follows:

```
$ git fetch
Receiving objects: 100% (9/9), done.
Resolving deltas: 100% (2/2), done.
From /path/to/repo/offline-sharing/myrepo.bundle
 * [new branch]      develop    -> origin/develop
```

We can now checkout, `develop`, and verify that the history for the `develop` and `master` branch matches the one in the original repository:

```
$ git checkout develop
Branch develop set up to track remote branch develop from
origin by rebasing.
Switched to a new branch 'develop'
$ gitk --all
```

The previous command gives the following output:

```
gitk: offline-other
SHA1 ID: c131c8bb2bf8254e46c013bfb33f4a61f9d4b40e
```

There's more...

The bundle is useful to update the history for repositories on machines where the normal transport mechanisms can't be used due to missing network connections between the machines, firewall rules, and so on. There are, of course, other methods than the Git bundle to transport the history to remote machines. A bare repository on a USB stick could also be used, or even plain patches can be applied to the repository. The advantage of the Git bundle is that you don't have to write the entire history to a bare repository each time you need to update a remote, but only the part of history that is missing.

Creating archives from a tree

Sometimes, it is useful to have a snapshot of the directory structure as specified by a particular commit, but without the corresponding history. This can, of course, be done by checking the particular commit followed by deleting/omitting the `.git` folder when creating an archive. But with Git, there is a better way to do this, which is built in so it is possible to create an archive from a particular commit or reference. When using Git to create the archive, you also make sure that the archive only contains the files tracked by Git and not any untracked files or folders you might have in your working directory.

Getting ready

We'll use the same `offline-sharing` repository as used in the previous examples in this chapter:

```
$ git clone https://github.com/dvaske/offline-sharing.git
Cloning into 'offline-sharing'...
remote: Counting objects: 32, done.
remote: Compressing objects: 100% (25/25), done.
remote: Total 32 (delta 7), reused 30 (delta 6)
Unpacking objects: 100% (32/32), done.
Checking connectivity... done.
$ cd offline-sharing
```

How to do it...

We'll start by creating an archive of the directory structure on the latest commit on the `master` branch. The `offline-sharing` repository is checked out on the `develop` branch by default, so we'll use the reference `origin/master` to specify the ref for the archive:

```
$ git archive --prefix=offline/ -o offline.zip origin/master
```

The `--prefix` option prepends the specified prefix to each file in the archive, effectively adding an `offline` directory as a root directory for the files in the repository, and the `-o` option tells Git to create the archive

in the `offline.zip` file which of course, is compressed in the zip format. We can investigate the zip archive to check whether the files contain the following:

```
$ unzip -l offline.zip
Archive: offline.zip
1e42a2dfa3a377d412efd27a77b973c75935c62a
      Length      Date    Time     Name
-----  -----
        0  04-10-14 09:19  offline/
     162  04-10-14 09:19  offline/README.md
     485  04-10-14 09:19  offline/another_pi.c
     672  04-10-14 09:19  offline/math.c
-----
      1319
                           4 files
```

If we look in the Git repository in the `origin/master` commit, we can see that the files are the same; the `-l` option tells Git to specify each file's size as follows:

```
$ git ls-tree -l origin/master
100644 blob c79cad47938a25888a699142ab3cdf764dc99193 162
 README.md
100644 blob 86df41b3a8bbfb588e57c7b27742cf312ab3a12a 485
 another_pi.c
100644 blob d393b41eb14561e583f1b049db716e35cef326c3 672
 math.c
```

There's more...

The archive command can also be used to create an archive for a subdirectory of the repository. We can use this on the `doc` branch of the repository to zip the content of the `Documentation` folder:

```
$ git archive --prefix=docs/ -o docs.zip
origin/doc:Documentation
```

Again, we can list the contents of the zip file and the `Documentation` tree at `origin/doc` as follows:

```
$ unzip -l docs.zip
Archive: docs.zip
      Length      Date    Time     Name
-----  -----

```

```

----- ----- ----- -----
 0 04-13-14 21:14 docs/
 99 04-13-14 21:14 docs/README.md
152 04-13-14 21:14 docs/build.md
-----
251          3 files
$ git ls-tree -l origin/doc:Documentation
100644 blob b65b4fc78c0e39b3ff8ea549b7430654d413159f 99
 README.md
100644 blob f91777f3e600db73c3ee7b05ea1b7d42efd8881 152
 build.md

```

There are other format options besides the zip format for the archive, for example, tar, tar.gz, and so on. The format can be specified with the `--format=<format>` option or as a suffix to the output file name with the `-o` option. The following two commands will produce the same output file:

```
$ git archive --format=tar.gz HEAD > offline.tar.gz
$ git archive -o offline.tar.gz HEAD
```

The Git archive command behaves a bit differently if a commit/tag ID or a tree ID is passed as an identifier. If a commit or tag ID is given, the ID will be stored in a global extended pax header for the tar format and as a file comment for the zip format. If only the tree ID is given, no extra information will be stored. You can actually see this in the previous examples where the first ID was given a branch as a reference. As the branch points to a commit, the ID of this commit was written as a comment to the file and we can actually see it in the output of the archive listing:

```
$ unzip -l offline.zip
Archive: offline.zip
1e42a2dfa3a377d412efd27a77b973c75935c62a
      Length      Date    Time     Name
-----      ----      ----
        0 04-10-14 09:19  offline/
    162 04-10-14 09:19  offline/README.md
    485 04-10-14 09:19  offline/another_pi.c
    672 04-10-14 09:19  offline/math.c
-----
      1319          4 files
```

In the second example, we also passed a branch as a reference, but furthermore we specified the Documentation folder as the subfolder we wanted to create an archive from. This corresponds to passing the ID of the tree to the archive command; hence, no extra information will be stored in the archive.

Chapter 11. Git Plumbing and Attributes

In this chapter, we will cover the following topics:

- Displaying the repository information
- Displaying the tree information
- Displaying the file information
- Writing a blob object to the database
- Writing a tree object to the database
- Writing a commit object to the database
- Keyword expansion with attribute filters
- Metadata diff of binary files
- Storing binaries elsewhere
- Checking the attributes of a file
- Attributes for exporting an archive

Introduction

Git distinguishes between porcelain commands and plumbing commands. Porcelain commands are the ones you will normally use as `add`, `commit`, `checkout`, `rebase`, `merge`, and so on. Plumbing commands are all the helper functions that execute the low-level work. If you run `git help` in a terminal, you'll get a list of approximately 20 commands—the normal porcelain commands. You can also list all the Git commands with the `-a`, `--all` option; this results in about 150 commands.

In the previous chapters of the book, we already used some Git plumbing commands, but without much explanation. In this chapter, we'll take a closer look at some of the more useful commands to display information about files and trees in the repository. We'll also learn how we can create history without the use of the `add` and `commit` Git porcelain commands. Finally, we'll look into another area of Git: attributes. We'll see how we can replace keywords or strings in files on `add` or `checkout`, how we can diff binary files using textual metadata,

and how we can transparently store binary files outside the repository, though added with `git add`. We'll see how to check the attributes of files in the repository and how we can use attributes while exporting our repository with `git archive`.

Displaying the repository information

It is fairly common to have some scripts that use repository information, for example, builds or release note generation. This small example will show some examples of the `rev-parse` command that can be very useful for scripting.

Getting ready

Clone the `data-model` repository from [Chapter 1, Navigating Git](#):

```
$ git clone https://github.com/dvaske/data-model.git  
$ cd data-model
```

How to do it...

First, let's figure out the ID of the commit at `HEAD`:

```
$ git rev-parse HEAD  
34acc370b4d6ae53f051255680feaefaf7f7850d
```

This can, of course, also be obtained by `git log -1 --format=%H`, but with the `rev-parse` command, you don't need all the options. We can also get the current branch from the `rev-parse` command:

```
$ git rev-parse --symbolic-full-name HEAD  
refs/heads/master
```

We can also just get the abbreviated name:

```
$ git rev-parse --symbolic-full-name --abbrev-ref HEAD  
master
```

We can also get the ID of other refs:

```
$ git rev-parse origin/feature/2  
82cd5662900a50063ff4bb790539fe4a6d470d56
```

There's more...

The `rev-parse` command can also be used to give some information about the repository, especially the directory structure relative to the current working directory.

We can find the top-level directory from a subdirectory using the following snippet:

```
$ cd a_sub_directory  
$ pwd  
/path/to/repo/data-model/a_sub_directory  
$ git rev-parse --show-toplevel  
/path/to/repo/data-model
```

We can also get the number of `cd ..` we need to get to the root directory using the following command:

```
$ git rev-parse --show-cdup  
../
```

And we can get the relative path from the root directory to the current working directory using the following command:

```
$ git rev-parse --show-prefix  
a_sub_directory/
```

Finally, we can get Git to show the path to the `.git` directory:

```
$ git rev-parse --git-dir  
/path/to/repo/data-model/.git
```

We can check whether the current repository is a bare repository using the following command:

```
$ git rev-parse --is-bare-repository  
false
```

Displaying the tree information

It can sometimes be useful to show or find information on certain trees and files in Git. Here, the `ls-tree` and `diff-tree` commands come in handy. Essentially, these are plumbing commands, and they can sometimes be very useful when scripting or browsing.

Getting ready

We'll use the same repository as the previous example:

```
$ git clone https://github.com/dvaske/data-model.git
```

We also need a `bare` repository for some examples; it will by default, with the `--bare` option, be cloned to `data-model.git`, so remember which repository you are currently working on:

```
$ git clone --bare https://github.com/dvaske/data-model.git
$ cd data-model
```

How to do it...

To show the content of the current tree in the Git context, we can use the `ls-tree` command. We'll pass `--abbrev` to the command to limit the SHA-1 ID to its abbreviated form. This is usually seven characters:

```
$ git ls-tree --abbrev HEAD
100644 blob f21dc28 README.md
040000 tree abc267d a_sub_directory
100644 blob b50f80a another-file.txt
100644 blob 92f046f cat-me.txt
100644 blob bb2fe94 hello_world.c
```

The columns in the output are as follows:

- **Mode:** In mode, `tree` is `040000`, a regular file is `100644`, and an executable file is `100755`
- **Type:** This is the object type `tree` or `blob`
- **SHA-1 ID:** This is the object identifier, that is, the SHA-1 of the

object

- **Path:** This is the file or the directory name

It's also possible to print the sizes of blobs in a column between the ID and the filename using the `-l`, `--long` option as follows:

```
$ git ls-tree --abbrev -l HEAD
100644 blob f21dc28      312  README.md
040000 tree abc267d      -  a_sub_directory
100644 blob b50f80a      26   another-file.txt
100644 blob 92f046f      78   cat-me.txt
100644 blob bb2fe94      101  hello_world.c
```

It is also possible to recursively list all the files with the `ls-tree` command:

```
$ git ls-tree --abbrev -r HEAD
100644 blob f21dc28  README.md
100644 blob 6dc3fb  a_sub_directory/readme
100644 blob b50f80a another-file.txt
100644 blob 92f046f cat-me.txt
100644 blob bb2fe94 hello_world.c
```

By default, the recursive option for `ls-tree` doesn't print tree objects, only blobs. To get this information as well while recursively listing the contents, pass the `-t` option as shown in the following command:

```
$ git ls-tree --abbrev -r -t HEAD
100644 blob f21dc28  README.md
040000 tree abc267d  a_sub_directory
100644 blob 6dc3fb  a_sub_directory/readme
100644 blob b50f80a another-file.txt
100644 blob 92f046f cat-me.txt
100644 blob bb2fe94 hello_world.c
```

The `ls-tree` command makes it easy to find a file in a repository, normal or bare. If for example, you need a single file in a bare repository and don't want to clone it, you can run `ls-tree` on the project and `grep` for the filename you need, or supply the path for the file if you know it. With the SHA-1 ID of the file, you can display the contents of the file, with either `cat-file` or `show`, and redirect the output to a file. Let's see an example of this; we're looking for the `readme` file in the

`a_sub_directory` directory. We'll use the bare repository, `data-model.git`, we created in the *Getting started* section of this recipe:

```
$ cd data-model.git
$ git ls-tree -r HEAD a_sub_directory/readme
100644 blob 6dc3bfbc6db8253b7789af1dee44caf8ec6ffb6e
a_sub_directory/readme
$ git cat-file -p 6dc3bfbc6db8253b7789af1dee44caf8ec6ffb6e
A file in a sub directory
```

We can also perform diffs of trees and see the result in the same type of view as the `ls-tree` command. Switch back to the normal repository, and we'll investigate the `diff-tree` command:

```
$ cd ..
$ cd data-model
```

Start by performing a diff of the `HEAD` commit with the `feature/2` branch in the `data-model` repo:

```
$ git diff-tree --abbrev HEAD origin/feature/2
:000000 100644 000000... 07ec769... A HelloWorld.java
:100644 100644 f21dc28... c4e7c18... M README.md
:040000 000000 abc267d... 0000000... D a_sub_directory
:100644 000000 b50f80a... 0000000... D another-file.txt
:100644 000000 92f046f... 0000000... D cat-me.txt
:100644 000000 bb2fe94... 0000000... D hello_world.c
:000000 100755 0000000... cf5edaa... A hello_world.pl
:000000 100755 0000000... 2bec0da... A hello_world.py
```

The output here differs from the output of `ls-tree`. There are two columns for mode and ID, one for each side of the diff. In this case, the branches are `master` and `origin/feature/2`. The second last column describes the state of the file in diff, `A` = added, `D` = deleted, `M` = modified, and `R` = renamed. If we just want to see the diff in files, without the mode and ID, we can specify the `--name-status` or the `--name-only` option:

```
$ git diff-tree --name-status HEAD origin/feature/2
A HelloWorld.java
M README.md
D a_sub_directory
D another-file.txt
```

```
D  cat-me.txt
D  hello_world.c
A  hello_world.pl
A  hello_world.py
$ git diff-tree --name-only HEAD origin/feature/2
HelloWorld.java
README.md
a_sub_directory
another-file.txt
cat-me.txt
hello_world.c
hello_world.pl
hello_world.py
```

The specific output format of the `ls-tree` and `diff-tree` commands makes them very useful for scripting in the cases where the file or tree information is needed.

Displaying the file information

While `git ls-tree` can give information about tree objects in the repository, it can't be used to display information about the index and working area. The `git ls-files` command can do this, and we will explore this next.

Getting ready

Again, we'll use the `data-model` repository from the previous example.

How to do it...

For specific file information, we can use the `ls-files` command to get information about files in the working tree and the staging area. By default, `git ls-files` will show the files in the staging area, as shown in the following command:

```
$ git ls-files
README.md
a_sub_directory/readme
another-file.txt
cat-me.txt
hello_world.c
```

Note this includes all files in the staging area (the tree state at the latest commit, `HEAD`) and files added since. The `status` command shows the changes between `HEAD` and staging area and the staging area and working tree. We can try to create a new file and see what happens using the following command:

```
$ echo 'Just testing...' > test.txt
$ git ls-files
README.md
a_sub_directory/readme
another-file.txt
cat-me.txt
hello_world.c
```

If we add the `test.txt` file to the staging area, we get the following output:

```
$ git add test.txt
$ git ls-files
README.md
a_sub_directory/readme
another-file.txt
cat-me.txt
hello_world.c
test.txt
```

By default, `ls-files` shows only the current content of the staging area, but let's see some other options. We can filter the output to only show modified files with the `-m`, `--modified` option, as shown in the following command:

```
$ echo "Another fine line" >> another-file.txt # Update tracked
file
$ git ls-files -m
another-file.txt
```

The `ls-files` command can also show files not tracked by Git but that exist in the work area. If we can remove the `test.txt` file from the staging area, the file is untracked and we can see that it shows up in the `ls-files` output with the `-o`, `--others` option:

```
$ git reset HEAD test.txt
$ git ls-files --others
test.txt
```

The `ls-files` command can also be used to show which files are ignored by Git with the `-i`, `--ignored` option. So, we'll add an exclude pattern on `.txt` files to `.gitignore` and check the output of `ls-files` with the standard exclude patterns (`.gitignore` in each directory, `.git/info/exclude`, and `~/.gitignore`):

```
$ echo '*.*' >> .gitignore
$ git ls-files -i --exclude-standard
another-file.txt
cat-me.txt
```

This showed two files already tracked by Git that match the exclude pattern, probably not really what we wanted. Let's try it again but this time on untracked files with the `-o`, `--others` option:

```
$ git ls-files -o -i --exclude-standard  
test.txt
```

This matches, as expected, the untracked files in the working directory, which are ignored in the `.gitignore` file.

Let's clean up for the next example:

```
$ git reset --hard origin/master  
$ git clean -xfd
```

There's more...

The `ls-files` command can also come in handy when performing merges that result in conflicts. With the `--unmerged` option, the output will only show files that have merge conflicts that haven't been fixed (added). To explore this, we can merge the `feature/2` branch into `master`:

```
$ git merge origin/feature/2  
Auto-merging README.md  
CONFLICT (content): Merge conflict in README.md  
Recorded preimage for 'README.md'  
Automatic merge failed; fix conflicts and then commit the  
result.
```

We can check `git status` to see which files have conflicts:

```
$ git status  
On branch master  
Your branch is up-to-date with 'origin/master'.  
You have unmerged paths.  
  (fix conflicts and run "git commit")
```

Changes to be committed:

```
new file:  HelloWorld.java  
new file:  hello_world.pl  
new file:  hello_world.py
```

```
Unmerged paths:
```

```
(use "git add <file>..." to mark resolution)
```

```
both modified: README.md
```

Let's see what the `--unmerged` option gives in the output:

```
$ git ls-files --unmerged
100644 23493f6b1edfafbee958d6daca863b1156850b7 1 README.md
100644 f21dc2804e888fee6014d7e5b1ceee533b222c15 2 README.md
100644 c4e7c18f3b31cd1d6fabeda4d7b009e3b4f12edb 3 README.md
```

As expected, the output only shows the conflicted file, `README.md`, but in three versions. The output is pretty similar to that of `ls-tree`, which we saw earlier; the first column and second column represent the file mode and the `SHA-1` ID, respectively. The third column is different though and is the stage number of the object. Here stage 0 represents the current working tree (not shown previously), but when a conflict arises, the stage number is used to distinguish between the different versions. Stage 1 represents the file where the other differences derive from, that is, the merge-base for that file. Stage 2 and 3 represent the same files, but with different revisions, as they were in their respective branches before the merge. If we want, we can check the revisions of the file with the `cat-file` or the `show` command. The `--unmerged` option also makes it easy to get a list of unmerged files when filtered a bit:

```
$ git ls-files --unmerged | cut -f 2 | sort -u
README.md
```

The output is much cleaner than `git status`, which also shows possible content in the staging area and untracked files.

Writing a blob object to the database

In this example, we'll see how we can use plumbing commands to add blob objects to the database. This is, of course, used internally by the `git add` command; however, this can also be useful if you, for example, need to add the public part of your GPG key to the Git repository so that signed tags can be verified. You can then, after you've added the `key`, tag the blob ID so that the other committers can find it.

Getting ready

We'll create and use a new repository for this example and the next couple of examples. Let's create a new repository in the `myplumbing` folder:

```
$ git init myplumbing
Initialized empty Git repository in /path/to/myplumbing/.git/
$ cd myplumbing
$ git status
On branch master

Initial commit

nothing to commit (create/copy files and use "git add" to track)
```

How to do it...

Git uses the `hash-object` plumbing command to write objects to its database. We can use it to update the database with the content of files, or pass the content directly on `stdin`. First, let's just see what is currently stored in the database (`.git/objects`) using the following command:

```
$ find .git/objects
.git/objects
.git/objects/info
```

```
.git/objects/pack  
$ find .git/objects -type f
```

The database is empty as expected when we have just created the repository. Now, we can write an object to the database:

```
echo 'This is the content of my file' | git hash-object -w --  
stdin  
70bacd9f51c26d602f474bbdc9f60644aa449e97
```

We can also try to use the `hash-object` command on a regular file:

```
$ echo 'This content is good' > mytest.txt  
$ git hash-object -w mytest.txt  
926e8ffd3258ed6edd1e254438f02fd24e417acc
```

We can update the file and write the new content to the database:

```
$ echo 'This content is better' > mytest.txt  
$ git hash-object -w mytest.txt  
6b3da706d14c3820597ec7109f163bc144dcbb22
```

How it works...

The `hash-object` function will create a SHA-1 hash of the input given and if the `-w` switch is used, write it to the database. The command defaults to blob objects. We can investigate the contents of the database using the following command:

```
find .git/objects -type f  
.git/objects/6b/3da706d14c3820597ec7109f163bc144dcbb22  
.git/objects/70/bacd9f51c26d602f474bbdc9f60644aa449e97  
.git/objects/92/6e8ffd3258ed6edd1e254438f02fd24e417acc
```

We can see that the database contains the three objects we just created. As you can see, Git stores each object as a file where the two first digits of the SHA-1 are used as a subdirectory and the remaining 38 objects as the filename. To check the object in Git's database, we can use the `cat-file` command, just like we did in [Chapter 1, Navigating Git](#). To check the contents of an object, we use the `-p` (pretty print) switch:

```
git cat-file -p 70bacd9f51c26d602f474bbdc9f60644aa449e97
```

This is the content of my file

We can also use the `cat-file` command to check the type of an object:

```
git cat-file -t 926e8ffd3258ed6edd1e254438f02fd24e417acc  
blob
```

There's more...

We can use the `cat-file` command to update files in our working directory, so to revert `mytest.txt` to the first version, use the following command:

```
$ git cat-file -p 926e8ffd3258ed6edd1e254438f02fd24e417acc >  
mytest.txt  
$ cat mytest.txt  
This content is good
```

Writing a tree object to the database

Now, we have manually created objects in the Git database, but as nothing point to these objects, we have to remember them by their SHA-1 identifier. Furthermore, only the content of the files is stored in the database, so we learn to create a tree object that will refer to the blobs created.

Getting ready

We'll use the same repository of the last examples with the objects we created in the database.

How to do it...

We'll start by adding the first version of `mytest.txt` as follows:

```
$ git update-index --add --cacheinfo 100644 \
926e8ffd3258ed6edd1e254438f02fd24e417acc mytest.txt
```

Now we can write the content of the staging area to the database:

```
$ git write-tree
4c4493f8029d491d280695e263e24772ab6962ce
```

We can update and write a tree for the second version of `mytest.txt` as follows:

```
$ git update-index --cacheinfo 100644 \
6b3da706d14c3820597ec7109f163bc144dcbb22 mytest.txt
$ git write-tree
2b9697438318f3a62a5e85d14a3b52d69b962907
```

Finally, we can use the object we created from `stdin` and we'll put it in a subdirectory, `sub`, with the name `other.txt`:

```
$ git update-index --add --cacheinfo 100644 \
70bacd9f51c26d602f474bbdc9f60644aa449e97 sub/other.txt
```

```
$ git write-tree  
9387b1a7619c6d899d83fe8d1437864bcd88736c
```

How it works...

The `update-index` command updates the staging area with the files or content specified. The `--add` command tells Git to add a new file to the index, and the `--cacheinfo` switch is used to tell the command to use an existing object from the database. The `100644` command is used to create a normal file. Other types are `100755` (the executable file) and `040000` (a subdirectory), and so on. We could also have used a file directly for the `update-index` command, and then just provided the filename and omitted `--cacheinfo`, `filetype`, and `identifier`.

We can check the objects created with the `cat-file` command and verify their types, as shown in the following snippet:

```
$ git cat-file -p 4c4493f8029d491d280695e263e24772ab6962ce  
100644 blob 926e8ffd3258ed6edd1e254438f02fd24e417acc  
mytest.txt  
$ git cat-file -t 4c4493f8029d491d280695e263e24772ab6962ce  
tree  
$ git cat-file -p 2b9697438318f3a62a5e85d14a3b52d69b962907  
100644 blob 6b3da706d14c3820597ec7109f163bc144dcbb22  
mytest.txt  
$ git cat-file -t 2b9697438318f3a62a5e85d14a3b52d69b962907  
tree  
$ git cat-file -p 9387b1a7619c6d899d83fe8d1437864bcd88736c  
100644 blob 6b3da706d14c3820597ec7109f163bc144dcbb22  
mytest.txt  
040000 tree 4cba806ece93e483f2515a7cc7326e194844797 sub  
$ git cat-file -t 9387b1a7619c6d899d83fe8d1437864bcd88736c  
tree
```

Writing a commit object to the database

Now that we have created both blob and tree objects, the next step in the data model is to create the actual commit object.

Getting ready

Again, we'll use the repository created in the previous examples with the different objects written to the database.

How to do it...

As we saw in [Chapter 1, Navigating Git](#), a commit object consists of the author and committer information, a root tree object, a parent commit (except for the first commit), and a commit message. We have the root tree object generated in the last example, and Git will pick up the author and committer information from the configuration. So, all we need to do is create a commit message and write the commit object. We can do this for each of the tree objects we created previously:

```
$ echo 'Initial commit - Good contents' | git commit-tree  
4c4493f8  
40f4783c37e7cb9d07a4a71100acf4c474a376b0  
$ echo 'Second commit - Better contents' | git commit-tree -p \  
40f4783 2b969743  
991ad244c6fdc84a983543cd8f2e89deca0eff29  
$ echo 'Adds a subdirectory' | git commit-tree -p 991ad244  
9387b1a7  
e89518224a971df09a00242355b62278964d6811
```

How it works...

Three commit objects are created. The `-p` switch in the latter two commands tells Git to use the commit specified as a parent commit for the one to be created. Git will use the author and committer information it can find through the configuration options. We can verify that the

commits were created with the `cat-file` command, just like we did with the blobs and trees:

```
$ git cat-file -p 40f4783c37e7cb9d07a4a71100acf4c474a376b0
tree 4c4493f8029d491d280695e263e24772ab6962ce
author Aske Olsson <aske.olsson@switch-gears.dk> 1398270736
+0200
committer Aske Olsson <aske.olsson@switch-gears.dk> 1398270736
+0200
```

Initial commit - Good contents

```
$ git cat-file -t 40f4783c37e7cb9d07a4a71100acf4c474a376b0
commit
$ git cat-file -p 991ad244c6fdc84a983543cd8f2e89deca0eff29
tree 2b9697438318f3a62a5e85d14a3b52d69b962907
parent 40f4783c37e7cb9d07a4a71100acf4c474a376b0
author Aske Olsson <aske.olsson@switch-gears.dk> 1398270736
+0200
committer Aske Olsson <aske.olsson@switch-gears.dk> 1398270736
+0200
```

Second commit - Better contents

```
$ git cat-file -t 991ad244c6fdc84a983543cd8f2e89deca0eff29
commit
$ git cat-file -p e89518224a971df09a00242355b62278964d6811
tree 5c23c103aeaa360342f36fe13a673fa473f665b8
parent 991ad244c6fdc84a983543cd8f2e89deca0eff29
author Aske Olsson <aske.olsson@switch-gears.dk> 1398270736
+0200
committer Aske Olsson <aske.olsson@switch-gears.dk> 1398270736
+0200
```

Adds a subdirectory

```
$ git cat-file -t e89518224a971df09a00242355b62278964d6811
commit
```

As we specified, parent commits for the last two commit objects we made we actually created Git's history in the repository, which we can view with the `log` command. We'll need to tell the `log` command to show the history from the latest commit, as we haven't updated any branches to point to it:

```
$ git log e89518224a971df09a00242355b62278964d6811
commit e89518224a971df09a00242355b62278964d6811
```

Author: Aske Olsson <aske.olsson@switch-gears.dk>

Date: Wed Apr 23 18:32:16 2014 +0200

Adds a subdirectory

commit 991ad244c6fdc84a983543cd8f2e89deca0eff29

Author: Aske Olsson <aske.olsson@switch-gears.dk>

Date: Wed Apr 23 18:32:16 2014 +0200

Second commit - Better contents

commit 40f4783c37e7cb9d07a4a71100acf4c474a376b0

Author: Aske Olsson <aske.olsson@switch-gears.dk>

Date: Wed Apr 23 18:32:16 2014 +0200

Initial commit - Good contents

Keyword expansion with attribute filters

With version control systems such as Subversion, RCS, and so on, it is possible to insert a number of keywords in a file, and these will then be expanded/collapsed on checkout/add. The same functionality can be achieved with Git, though a bit of setting up is required, as it is not built into Git's core. The attributes functionality of Git can be used to create filters for the keyword substitution. In the following example, we'll see how we can easily exchange the `$Date$` keyword with a string such as `$Date: Sun Apr 27 14:17:24 2014 +0200$` on checkout.

Getting ready

In this example, we'll use the repository located at https://github.com/dvaske/attributes_example.git. Take a look at the keyword branch using the following command:

```
$ git clone https://github.com/dvaske/attributes_example.git
$ cd attributes_example
$ git checkout keyword
```

How to do it...

First, let's create the filters needed to substitute the keyword on `add` and `checkout`. The filters are in Git called **clean** and **smudge**. The clean filter runs on `add` (check-in) to make sure whatever is added to Git is cleaned. The smudge filter runs on `checkout` and can expand all the keywords in the file, smudging the file. Configure the clean filter for Git (local to this repository):

```
git config filter.date-keyword.clean 'perl -pe \
"s/\\"\\\$Date[^\\"\\\$]*\\\$/\\"\\\$Date\\\$/"'
```

Configure the clean filter for Git (local to this repository):

```
git config filter.date-keyword.smudge 'perl -pe \

```

```
"s/\$\Date[^$]*$/$\Date: 'git log -1 --all --  
format=%ad \  
-- $1'\"$/"
```

With the filters configured, we now need to tell Git which files to run those filters on. We will write this information in the `.gitattributes` file in the root of the repository. We'll add any `c` or `java` file affected by these filters:

```
echo "*.c filter=date-keyword" > .gitattributes  
echo "*.java filter=date-keyword" >> .gitattributes
```

Now the content of any `c` or `java` file will be passed through the filters on add and checkout.

How it works...

We can check how it works by investigating the files in the workspace. Let's take a closer look at `HelloWorld.java`:

```
$ cat HelloWorld.java  
/* $Date$ */  
  
public class HelloWorld {  
    public static void main(String[] args) {  
        System.out.println("Hello, world!");  
    }  
}
```

Nothing has happened to the file, though the filters are in place, as the filters only run on add/checkout. We can delete the file and check it out again to see the filters in action:

```
$ rm HelloWorld.java  
$ git checkout HelloWorld.java  
$ cat HelloWorld.java  
/* $Date: Sun Apr 27 14:32:49 2014 +0200$ */  
  
public class HelloWorld {  
    public static void main(String[] args) {  
        System.out.println("Hello, world!");  
    }  
}
```

If we change `HelloWorld` in the two files and add them to the staging area, we can verify that the java file has been cleaned when added using the following command:

```
$ git add HelloWorld.java
$ git diff -U5 --cached
index 233cc49..3905c75 100644
--- a/HelloWorld.java
+++ b/HelloWorld.java
@@ -1,7 +1,7 @@
/* $Date$ */

public class HelloWorld {
    public static void main(String[] args) {
-        System.out.println("Hello, world!");
+        System.out.println("Hello again, world!");
    }
}
```

As expected, we can see that when added, the file has been cleaned of the date information.

There's more...

We can continue to change `hello world` in the `c` file, also to: `hello again world` and add the `c` file and the `.gitattributes` file to the staging area. Then, we can create a commit as follows:

```
$ git add hello_world.c
$ git add .gitattributes
$ git commit -m 'Add date-keyword filter for *.c and *.java
files'
[keyword 28a0009] Add date-keyword filter for *.c and *.java
files
 3 files changed, 4 insertions(+), 2 deletions(-)
 create mode 100644 .gitattributes
```

How does the `hello_world.c` file look in the working tree now? As we only changed a line in the file and added it afterwards, the file was never put through a checkout cycle, so we shouldn't expect the `$Date$` string to be expanded.

```
$ cat hello_world.c
```

```
/* $Date$ */

#include <stdio.h>

int main(void)
{
    printf("hello, world\n");
    return 0;
}
```

If we switch branch to `master` and then back to `keyword`, the files will have been through the checkout filter:

```
$ git checkout master
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.
$ git checkout keyword
Switched to branch 'keyword'
Your branch is ahead of 'origin/keyword' by 1 commit.
  (use "git push" to publish your local commits)
$ head -1 hello_world.c
/* $Date: Thu May 1 22:59:21 2014 +0200$ */
$ head -1 HelloWorld.java
/* $Date: Thu May 1 22:59:21 2014 +0200$ */
```

From the previous output, you can now see the `Date` keyword expanded to a timestamp.

Metadata diff of binary files

Binary files can be hard to diff, depending on the type of the file. Often, the only option is to load two instances of the program to show the files and check the differences visually. In this recipe we'll see how we can use EXIF metadata to diff images in the repository.

Getting ready

We'll use the same repository as we did in the last example and either re-clone it or checkout the `exif` branch:

```
$ git clone https://github.com/dvaske/attributes_example.git
$ cd attributes_example
$ git checkout exif
```

How to do it...

In order to use the EXIF data while diffing binary files, we need to set up a filter to tell Git what to do when a file of `*.jpg` is to be diffed. EXIF data is metadata embedded in images and is often used by digital cameras to record timestamps, the size of an image, and so on.

We'll write the following line to `.gitattributes`:

```
*.jpg diff=exif-diff
```

This only tells Git that JPG files should use the `exif-diff` filter; we still need to set it up. To extract the EXIF metadata, there are different programs such as `exiftool`, `jhead`, and so on. In this example, we're using `exiftool`, so make sure you have it installed and available on your `PATH`. To set up the `exiftool` diff filter, we create the following Git config:

```
git config diff.exif-diff.textconv exiftool
```

From now on, every time `jpg` is to be diffed, you'll just see a comparison of `exifdata`. To see the actual change in the image, you still have to

show the two images and visually compare them.

How it works...

Now that the filter is set up, we can try to check the output of it. The last two commits in the repository on the `exif` branch contain pictures that have had their size changed; let's see how they look with the `exif-diff` filter. First, check `log` for the last two commits:

```
$ git log --name-status -2
commit 0beb82c65d8cd667e1ffe61860a42a106be3c1a6
Author: Aske Olsson <aske.olsson@switch-gears.dk>
Date:   Sat May 3 14:55:50 2014 +0200
```

Changes sizes of images

```
M      europe_needles.jpg
M      hello_world.jpg
M      pic_credits.txt
```

```
commit a25d0defc70b9a1842463c1e9894a88dfb897cd8
Author: Aske Olsson <aske.olsson@switch-gears.dk>
Date:   Sun Apr 27 16:02:51 2014 +0200
```

Adds pictures to repository

Picture credits found in pic_credits.txt

```
M      README.md
A      europe_needles.jpg
A      hello_world.jpg
A      pic_credits.txt
```

Let's look at the diff between the two commits (the output you get might not match the following output 1:1 and depends on the `exiftool` version, OS, and so on. The following output is generated with `exiftool 9.61` on OS X 10.9.3):

```
$ git diff HEAD^..HEAD
diff --git a/europe_needles.jpg b/europe_needles.jpg
index 7291028..44e98e3 100644
--- a/europe_needles.jpg
+++ b/europe_needles.jpg
```

```

@@ -1,11 +1,11 @@
-ExifTool Version Number          : 9.54
-File Name                      : Gnepvw_europe_needles.jpg
-Directory                      :
/var/folders/3r/6f35b4t11rv2nmbrx5x32t4w0000gn/T
-File Size                       : 813 kB
+File Name                      : europe_needles.jpg
+Directory                      :
+File Size                       : .
-File Modification Date/Time    : 2014:05:03 22:08:05+02:00
-File Access Date/Time          : 2014:05:03 22:08:05+02:00
+File Access Date/Time          : 2014:05:03 22:08:06+02:00
-File Inode Change Date/Time   : 2014:05:03 22:08:05+02:00
-File Permissions               : rw-----
+File Permissions               : rw-r--r--
-File Type                       : JPEG
-MIME Type                      : image/jpeg
-JFIF Version                   : 1.01
@@ -79,8 +79,8 @@ Sub Sec Time Original      : 00
Sub Sec Time Digitized         : 00
Flashpix Version               : 0100
Color Space                     : sRGB
-Exif Image Width              : 1620
-Exif Image Height             : 1080
+Exif Image Width              : 1024
+Exif Image Height             : 683
...

```

There's more...

It is also possible to set up diffing of binary files for other types than images. As long as some useful data from the file type can be extracted, it's possible to create a custom diff filter for that file type. With the `catdoc` program, Microsoft Word files can, for example, be translated from the `.doc` format to plain text, which makes it easy to diff the text content in two files, but not their formatting.

Storing binaries elsewhere

Though binaries can't easily be diffed, there is nothing to prevent them from being stored in a Git repository, and there are no issues in doing so. However, if one or more binaries in a repository are updated frequently, it can cause the repository to grow quickly in size, making clones and updates slow as a lot of data needs to be transferred. By using the clean and smudge filters for the binaries, it is possible to move them to another location while adding them to Git and fetch them from that location while checking out the specific version of the file.

Getting ready

We'll use the same repositories as in the previous example, but the `no_binaries` branch:

```
$ git clone https://github.com/dvaske/attributes_example.git
$ cd attributes_example
$ git checkout no_binaries
```

How to do it...

First, we need to set up the clean and smudge filters for the files. Then, we are only going to run the filter on `.jpg` files in this example, so let's set it up and create the configuration:

```
$ echo '*.*.jpg filter=binstore' > .gitattributes
```

Create configuration

```
$ git config filter.binstore.clean "./put-bin"
$ git config filter.binstore.smudge "./get-bin"
```

We also need to create the actual filter logic, the `put-bin` and `get-bin` files, to handle the binary files on `add` and `checkout`. For this example, the implementation is very simple (no error handling, retries, and so on, is implemented).

The clean filter (to store the binaries somewhere else) is a simple bash script that stores the binary it receives on `stdin` to a directory called `binstore` at the same level as the Git repository. Git's own `hash-object` function is used to create a SHA-1 ID for the binary. The ID is used as filename for the binary in the `binstore` folder and is written as the output of the filer, as the content of the binary file when stored in Git.

The filter logic for the `put-bin` file can be created as follows:

```
#!/bin/bash
dest=$(git rev-parse --show-toplevel)/../binstore
mkdir -p $dest
tmpfile=$(git rev-parse --show-toplevel)/tmp
cat > $tmpfile
sha=$(git hash-object --no-filters $tmpfile)
mv $tmpfile $dest/$sha
echo $sha
```

The smudge filter fetches the binaries from the `binstore` storage on the same level as the Git repository. The content of the file stored in Git, the SHA-1 ID, is received on `stdin` and is used to output the content of the file by that name in the `binstore` folder:

The filter logic for the `get-bin` file can be created as follows:

```
#!/bin/bash
source=$(git rev-parse --show-toplevel)/../binstore
tmpfile=$(git rev-parse --show-toplevel)/tmp
cat > $tmpfile
sha=$(cat $tmpfile)
cat $source/$sha
rm $tmpfile
```

Create these two files and put them in the root of the Git repository.

Now, we are ready to add a JPG image to our repository and see that it is stored somewhere else. We can use the `hello_world.jpg` image from the `exif` branch. We can create the file here by querying Git. Find the SHA-1 ID of `hello_world.jpg` at the tip of the `exif` branch:

```
$ git ls-tree --abbrev exif | grep hello_world
```

```
100644 blob 5aac2df hello_world.jpg
```

Create the file by reading the content from Git to a new file:

```
$ git cat-file -p 5aac2df > hello_world.jpg
```

Now, we can add the file, commit the file, and check the external storage, which is placed relative to the current repository at `../binstore`, and see the commit content:

Add `hello_world.jpg` using the following command:

```
$ git add hello_world.jpg
```

Commit the contents of the staging area, the `hello_world.jpg` file:

```
$ git commit -m 'Added binary'  
[no_binaries 19e359d] Added binary  
 1 file changed, 1 insertion(+)  
 create mode 100644 hello_world.jpg
```

Check the content of the `binstore` directory:

```
$ ls -l ../binstore  
total 536  
-rw-r--r-- 1 aske staff 272509 May 3 23:24  
5aac2dff477eebb3da3cb68843b5cc39745d6447
```

Finally, we can check the content of the commit with the `-p` option to display the patch of the commit:

```
$ git log -1 -p  
commit 19e359d774c880fa4f37a3f41a874ba632a31c65  
Author: Aske Olsson <aske@schantz.com>  
Date:   Sat May 3 22:56:46 2014 +0200
```

```
Added binary
```

```
diff --git a/hello_world.jpg b/hello_world.jpg  
new file mode 100644  
index 000000..19680e5  
--- /dev/null  
+++ b/hello_world.jpg  
@@ -0,0 +1 @@
```

```
+5aac2dff477eebb3da3cb68843b5cc39745d6447
```

`hello_world.jpg` is a new file with `5aac2dff477eebb3da3cb68843b5cc39745d6447` content that is similar, as expected, to the name of the file in the `binstore` directory.

How it works...

Each time a `.jpg` file is added, the `put-bin` filter runs. The filter receives the content of the added file on `stdin`, and it has to output the result of the filter (what needs to go into Git) on `stdout`. The following is the filter explained in detail:

```
dest=$(git rev-parse --show-toplevel)/../binstore  
mkdir -p "$dest"
```

The previous two lines create the `binstore` directory if it doesn't exist. The directory is created at the same level as the Git repository:

```
tmpfile=$(git rev-parse --show-toplevel)/tmp  
cat > $tmpfile
```

The `tmpfile` variable is just a path to a temporary file, `tmp`, located in the root of the repository. The input received on `stdin` is written to this file.

```
sha=$(git hash-object --no-filters $tmpfile)  
mv $tmpfile $dest/$sha
```

The previous lines use Git's hashing function to generate a hash for the content of the binary file. We'll use the hash of the file as an identifier when we move it to the `binstore` folder where the SHA-1 will function as the filename of the binary.

```
echo $sha
```

Finally, we output the hash of the file to `stdout`, and this will be what Git stores as the content of the file in the Git database.

The smudge filter to populate our working tree with the correct file

contents also receives the content (from Git) on `stdin`. The filter needs to find the file in the `binstore` directory and write the content to `stdout` for Git to pick it up as the smudged file.

```
src=$(git rev-parse --show-toplevel)/../binstore  
tmpfile=$(git rev-parse --show-toplevel)/tmp  
cat > $tmpfile
```

The first three lines define the path to the `binstore` folder and a temporary file to which the content received from Git is written.

```
sha=$(cat $tmpfile)
```

The hash of the file we need to get is extracted in the previous line.

```
cat $src/$sha  
rm $tmpfile
```

Finally, we can output the real contents of the file to `stdout` and remove the temporary file.

There's more...

The previous filters work transparently with Git on `add` and `checkout`, but there are some caveats when using Git attributes, and especially filters like the previous ones, which are:

- Even though the `.gitattributes` file can be added and distributed inside the repository, the configuration of the filters can't. The configuration of the filters was the first step of the example, which tells Git which command to run for clean and smudge when the filter is used:

```
$ git config filter.binstore.clean "./put-bin"  
$ git config filter.binstore.smudge "./get-bin"
```

- The configuration can be either local to the repository, global for the user, or global for the system, as we saw in [Chapter 2](#), *Configuration*. However, none of these configurations can be distributed along with the repository, so it is very important that the configuration is set up just after clone. Otherwise, the risk of adding

- a file without running through the filters is too high.
- In this example, the storage location of the binaries is just a local directory next to the repository. A better way of doing this could be to copy the binaries to a central storage location either with, for example, `scp` or through a web service. This, however, limits the user from adding and committing when offline as the binaries cannot be stored in the central repository. A solution to this could be a `pre-push` hook that could transfer all the binaries to a binary database before a push happens.
- Finally, there is no error handling in the previous two filters. If one of them fails, it might make sense to abort the `add` or `checkout` and warn the user.

See also

There are also other ways of handling binaries in a repository that might be worth considering. These usually introduce extra commands to add and retrieve the binaries. The following are the examples of binary handlers:

- git-annex handler at <https://git-annex.branchable.com/>
- git-media handler at <https://github.com/schacon/git-media>
- git-bin handler at <https://github.com/Mighty-M/git-bin>

Checking the attributes of a file

Checking the `.gitattributes` file (or other places where attributes can be defined) to see whether a specific file is affected by an attribute can be quite cumbersome, especially if there are many entries in these files. Git has a built-in method that can be used to tell whether a file has any attribute associated.

Getting ready

We'll use the `attributes_example` repository:

```
$ git clone https://github.com/dvaske/attributes_example.git  
$ cd attributes_example
```

How to do it...

We'll start by setting up all the attributes we had in the last example:

```
$ echo '*.jpg filter=binstore' > .gitattributes  
$ echo '*.jpg diff=exif-diff' >> .gitattributes  
$ echo "*.c filter=date-keyword" >> .gitattributes  
$ echo "*.java filter=date-keyword" >> .gitattributes
```

Now we are ready to check different files. We'll start on the `keyword` branch and check the two code files using the following command:

```
$ git checkout keyword  
Branch keyword set up to track remote branch keyword from  
origin by rebasing.  
Switched to a new branch 'keyword'  
$ git check-attr -a hello_world.c HelloWorld.java  
hello_world.c: filter: date-keyword  
HelloWorld.java: filter: date-keyword
```

Let's also see the jpg files on the `exif` branch:

```
$ git checkout exif  
Branch exif set up to track remote branch exif from origin by  
rebasing.  
Switched to a new branch 'exif'
```

```
$ git check-attr -a hello_world.jpg europe_needles.jpg
hello_world.jpg: diff: exif-diff
hello_world.jpg: filter: binstore
europe_needles.jpg: diff: exif-diff
europe_needles.jpg: filter: binstore
```

It is also possible to check a file against a specific attribute with the following command:

```
$ git check-attr diff hello_world.jpg
hello_world.jpg: diff: exif-diff
$ git check-attr filter hello_world.jpg
hello_world.jpg: filter: binstore
```

If we check a file without attributes or against an attribute not associated with the file, the output is empty:

```
$ git check-attr -a README.md
$
```

Attributes to export an archive

While exporting a snapshot of a Git repository with the archive command (refer to [Chapter 10, Patching and Offline Sharing](#)), it is possible to change the way the archive is made.

Getting ready

We'll use the `attributes_example` repository:

```
$ git clone https://github.com/dvaske/attributes_example.git
$ cd attributes_example
```

How to do it...

First, we'll set up the attributes needed in `.gitattributes` and commit the file on the `exif` branch:

```
$ git checkout exif
Branch exif set up to track remote branch exif from origin by
rebasing.
Switched to a new branch 'exif'
$ echo 'europe_needles.jpg export-ignore' >> .gitattributes
$ git add .gitattributes
$ git commit -m 'Add .gitattributes'
[exif 783b7f7] Add .gitattributes
 1 file changed, 1 insertion(+)
 create mode 100644 .gitattributes
```

Now, we can create an archive from the tip of the `exif` branch, and the `europe_needles.jpg` file shouldn't be included, as shown in the following snippet:

```
$ git archive -o attr.zip exif
$ unzip -l attr.zip
Archive: attr.zip
783b7f73110e23f56675f0014ab3f1d0aba21d7f
 Length      Date    Time     Name
-----      ----   ----
      33  05-06-14 21:33  .gitattributes
     325  05-06-14 21:33  README.md
```

```
272509 05-06-14 21:33 hello_world.jpg
      543 05-06-14 21:33 pic_credits.txt
-----
273410                               4 files
```

The `europe_needles.jpg` file isn't there! This is very useful when creating archives of the source code without including test, the proprietary code, IPR, and so on.

There's more...

We can also do keyword substitution while exporting an archive. We need to use the `export-subst` attribute. We can set it up on the `README.md` file in the following way:

```
$ echo "README.md export-subst" >> .gitattributes
$ echo "Last commit: \$Format:%H\$" >> README.md
$ echo "Last commit date: \$Format:%cd\$" >> README.md
$ git add .gitattributes README.md
$ git commit -m "Commit info for git archive"
[exif 8c01a48] Commit info for git archive
 2 files changed, 3 insertions(+)
```

Create the archive. Check the content of the `README.md` file in the archive and the last commit on the `exif` branch using the following command:

```
$ git archive -o attr.zip exif
$ unzip -p attr.zip README.md
Git Attributes
=====
```

A few examples on using git attributes.

Pictures used found on flickr.com,
check `pic_credits.txt` for details.

Pictures only changes in size, not altered
only exif data is used as examples of diff'ing
pictures based on exif data.

Example repository for the book: Git Version Control Cookbook
Last commit: d3dda23601a3cc16295bdd7f4f9812544ea69d53

```
Last commit date: Tue May 6 21:52:25 2014 +0200 $  
$  
$ git log -1 --format=Commit: %H%nDate: %cd  
Commit: d3dda23601a3cc16295bdd7f4f9812544ea69d53  
Date: Tue May 6 21:52:25 2014 +0200
```

Recording the commit ID in the archive is very useful. This is especially the case if you are using the archive for testing so you know which revision or ID of the repository you are using, and you can report issues against this revision or ID.

Chapter 12. Tips and Tricks

In this chapter, we will cover the following topics:

- Using git stash
- Saving and applying stashes
- Debugging with git bisect
- Using the blame command
- Color UI in the prompt
- Autocompletion
- Bash prompt with status information
- More aliases
- Interactive add
- Interactive add with Git GUI
- Ignoring files
- Showing and cleaning ignored files

Introduction

In this chapter, you will find some tips and tricks that can be useful in everyday Git work. From stashing away your changes when you get interrupted with an important task over efficient bug hunting with `bisect` and `blame`, to color and status information in your prompt. We'll also look at aliases, how you can create clean commits by selecting which lines should be included in the commit, and finally how you can ignore files.

Using git stash

In this example, we explore the `git stash` command and learn how we can use it to quickly put away uncommitted changes and retrieve these again. This can be useful when being interrupted with an urgent task and you are not yet ready to commit the work you currently have in your working directory. With the `git stash` command, you save the state of your current working directory with/without a staging area and restore a clean state of the working tree.

Getting ready

In this example, we'll use the `cookbook-tips-tricks` repository. We'll use the `master` branch, but before we are ready to try the `stash` command, we need to create some changes in the working directory and the staging area:

```
$ git clone https://github.com/dvaske/cookbook-tips-tricks.git  
$ cd cookbook-tips-tricks  
$ git checkout master
```

Make some changes to `foo` and add them to the staging area:

```
$ echo "Just another unfinished line" >> foo  
$ git add foo
```

Make some changes to `bar` and create a new file:

```
$ echo "Another line" >> bar  
$ echo "Some content" > new_file  
$ 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: foo

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:   bar
```

```
Untracked files:
```

```
(use "git add <file>..." to include in what will be  
committed)
```

```
new_file
```

We can see that we have one file added to the staging area, `foo`, one modified file, `bar`, and an untracked file in the work area as well, `new_file`.

How to do it...

With the preceding state of our repository, we can stash away the changes so that we can work on something else. The basic command will put away changes from the staging area and changes made to the tracked files; it leaves untracked files in the working directory:

```
$ git stash  
Saved working directory and index state WIP on master: d611f06  
Update foo and bar  
HEAD is now at d611f06 Update foo and bar  
$ git status  
On branch master  
Your branch is up-to-date with 'origin/master'.
```

```
Untracked files:
```

```
(use "git add <file>..." to include in what will be  
committed)
```

```
new_file
```

```
nothing added to commit but untracked files present (use "git  
add" to track)
```

Now we can work on something else and create and commit this. We'll change the first line of the `foo` file and create a commit with this change:

```
$ sed -i '' 's/First line/This is the very first line of the
```

```
foo file/' foo
$ git add foo
$ git commit -m "Update foo"
[master fa4b595] Update foo
 1 file changed, 1 insertion(+), 1 deletion(-)
```

We can see the current work we have stashed away with the `git stash` list command:

```
$ git stash list
stash@{0}: WIP on master: 09156a4 Update foo and bar
```

To get back the changes we stashed away, we can pop them from the stash stack:

```
$ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)
```

Untracked files:

```
(use "git add <file>..." to include in what will be
committed)
```

```
new_file
nothing added to commit but untracked files present (use "git
add" to track)
$ git stash pop
Auto-merging foo
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)
```

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:   bar
modified:   foo
```

Untracked files:

```
(use "git add <file>..." to include in what will be
committed)
```

```

new_file
no changes added to commit (use "git add" and/or "git commit -a")
Dropped refs/stash@{0}
(91b68271c8968fed01032ad02322292f35be8830)

```

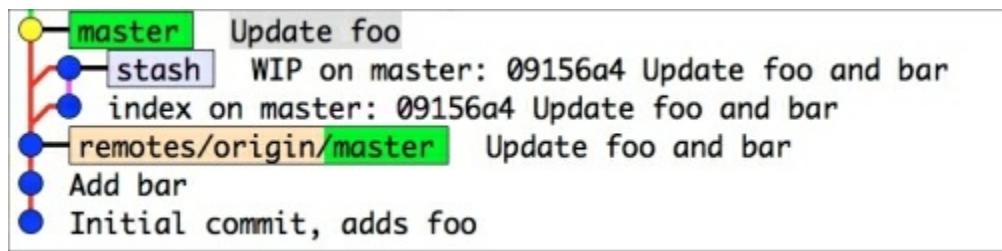
Now, the stashed changes are available again in the working repository and the stash entry is deleted. Note that the changes are applied only to the working directory, though one of the files was staged when we created the stash.

How it works...

We have created two commits: one for the index and one for the work area. In Gitk, we can see the commits that stash creates to put the changes away:



We can also see the state of the branches after we created the commit, as shown in the following screenshot:



Git actually creates two commits under the `refs/stash` namespace. One commit contains the content of the staging area. This commit is called `index on master`. The other commit is the work in progress in the working directory, `WIP on master`. When Git puts away the changes by

creating commits, it can use its normal resolution methods to apply the stashed changes back to the working directory. This means that if a conflict arises when applying the stash, it needs to be solved in the usual way.

There's more...

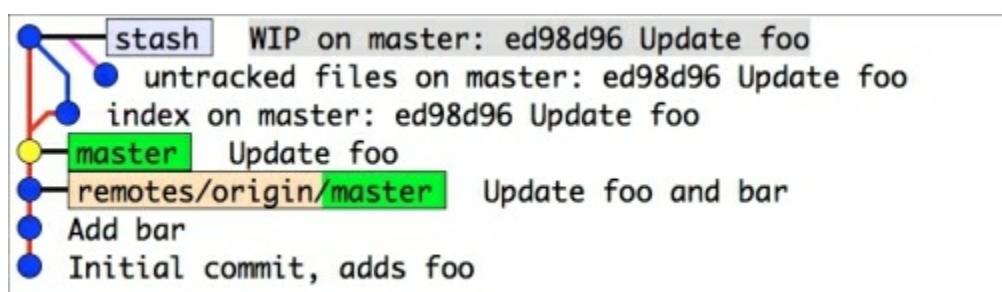
In the preceding example, we saw only the very basic usage of the `stash` command, putting away changes to untracked files and changes added to the staging area. It is also possible to include untracked files in the `stash` command. This can be done with the `--include-untracked` option. We can add `foo` to the staging area; firstly, to have the same state as when we created the stash earlier and then to create a stash that includes untracked files:

```
$ git add foo
$ git stash --include-untracked
Saved working directory and index state WIP on master: 691808e
Update foo
HEAD is now at 691808e Update foo
$ 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
```

Now, we can see that `new_file` has disappeared from the working directory. It is included in the stash and we can check this with Gitk. It will show up as another commit of untracked files:

```
$ gitk master stash
```



We can also make sure that the changes we added to the staging area are added back to the staging area after we apply the stash, so we end up with the exact same state as before we stashed away our changes:

```
$ git stash pop --index
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   foo

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:   bar

Untracked files:
  (use "git add <file>..." to include in what will be
committed)

    new_file

Dropped refs/stash@{0}
(ff331af57406948619b0671dab8b4f39dale8fa2)
```

It is also possible to only put away the changes in the working directory, keeping the changes in the staging area. We can do this either for only the tracked files or by stashing away untracked files (`--include-untracked`) as follows:

```
$ git stash --keep-index --include-untracked
Saved working directory and index state WIP on master: 00dd8f8
Update foo
HEAD is now at 00dd8f8 Update foo
$ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)
```

Changes to be committed:
(use "git reset HEAD <file>..." to unstage)

modified: foo

Saving and applying stashes

When stashing away work, we can easily have more than one state of work stashed away at a time. However, the default names for the stashed away changes aren't always helpful. In this example, we'll see how we can save stashes and name them so that it is easy to identify them again when listing the content of the stash. We'll also learn how to apply a stash without deleting it from the stash list.

Getting ready

We'll use the same repository as in the previous example, continuing from the state we left it in:

```
$ cd cookbook-tips-tricks
$ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   foo

$ git stash list
stash@{0}: WIP on master: 4447f69 Update foo
```

How to do it...

To save the current state to a stash with a description we can remember a later point in time, use the following command:

```
$ git stash save 'Updates to foo'
Saved working directory and index state On master: Updates to
foo
HEAD is now at 4447f69 Update foo
```

Our stash list now looks like the following:

```
$ git stash list
```

```
stash@{0}: On master: Updates to foo
stash@{1}: WIP on master: 2302181 Update foo
```

We can change `bar` and create a new stash:

```
echo "Another change" >> bar
$ git stash save 'Made another change to bar'
Saved working directory and index state On master: Made another
change to bar
HEAD is now at 2302181 Update foo
$ git stash list
stash@{0}: On master: Made another change to bar
stash@{1}: On master: Updates to foo
stash@{2}: WIP on master: 2302181 Update foo
```

We can apply the stashes back to the working tree (and staging area with the `--index` option) without deleting them from the stash list:

```
$ git stash apply 'stash@{1}'
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
(use "git push" to publish your local commits)
```

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:   foo

no changes added to commit (use "git add" and/or "git commit -a")
$ git stash apply --quiet 'stash@{0}'
$ git stash list
stash@{0}: On master: Made another change to bar
stash@{1}: On master: Updates to foo
stash@{2}: WIP on master: 2302181 Update foo
```

The stashes are still in the stash list, and they can be applied in any order and referred to with the `stash@{stash-no}` syntax. The `--quiet` option suppresses the status output after the stashes have been applied.

There's more...

For the stashes applied with `git stash apply`, the stash needs to be deleted with `git stash drop`:

```
$ git stash drop 'stash@{1}'  
Dropped stash@{1} (e634b347d04c13fc0a0d155a3c5893a1d3841fcd)  
$ git stash list  
stash@{0}: On master: Made another change to bar  
stash@{1}: WIP on master: 1676cdb Update foo
```

Keeping the stashes in the stash list by using `stash apply` and explicitly deleting them with `git stash drop` has some advantage over just using `stash pop`. When using the `pop` option, the stashes in the list are automatically deleted if they can be successfully applied, but if it fails and triggers the conflict resolution mode, the stash applied is not dropped from the list and continues to exist on the stash stack. This might later lead to accidentally using the wrong stash because it was thought to be gone. By consistently using `git stash apply` and `git stash drop`, you can avoid this scenario when done.

Tip

The `git stash` command can also be used to apply debug information to an application. Let's pretend you have been bug hunting and have added a lot of debug statements to your code in order to track down the bug. Instead of deleting all those debug statements, you can save them as a Git stash:

```
git stash save "Debug info stash"
```

Then, if you later need debug statements, you can just apply the stash and you are ready to debug.

Debugging with git bisect

The `git bisect` command is an excellent tool to find which commit caused a bug in the repository. The tool is particularly useful if you are looking at a long list of commits that may contain the bug. The `bisect` command performs a binary search through the commit history to find the commit that introduced the bug as fast as possible. The binary search method, or bisection method as it is also called, is a search method where an algorithm finds the position of a key in a sorted array. In each step of the algorithm, the key is compared to the middle value of the array and if they match, the position is returned. Otherwise, the algorithm repeats its search in the subarray to the right or left of the middle value, depending on whether the middle value was greater or lower than the key. In the Git context, the list of commits in the history makes up for the array of values to be tested, and the key can be a test if the code can be compiled successfully at the given commit. The binary search algorithm has a performance of $O(\log n)$.

Getting ready

We'll use the same repository as seen in the last example, but from a clean state:

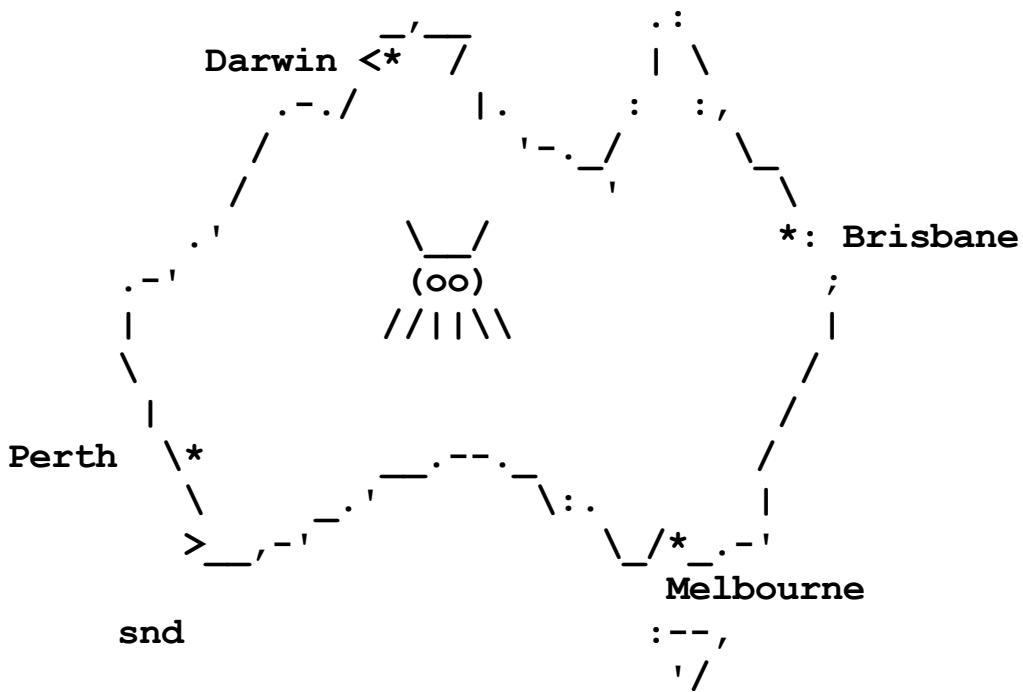
```
$ git clone https://github.com/dvaske/cookbook-tips-tricks.git
$ cd cookbook-tips-tricks
$ git checkout bug_hunting
```

The `bug_hunting` branch contains 23 commits since it branched off from the `master` branch. We know that the tip of the `bug_hunting` branch contains the bug and it was introduced in some commit since it branched off from `master`. The bug was introduced in the following commit:

```
commit 83c22a39955ec10ac1a2a5e7e69fe7ca354129af
Author: HAL 9000 <aske.olsson@switch-gears.dk>
Date:   Tue May 13 09:53:45 2014 +0200
```

Bugs...

The bug is easily seen in the `map.txt` file in the middle of Australia. The following snippet of the file shows the bug:



Now, all we need is some way to reproduce/detect the bug so we can test the different commits. This could, for example, simply be to compile the code, run tests, and so on. For this example, we'll create a test script to check for bugs in the code (a simple `grep` for `oo` should do it in this example; see for yourself if you can find the bug in the `map.txt` file):

```
echo "! grep -q oo map.txt" > ../test.sh  
chmod +x ../test.sh
```

It is best to create this test script outside the repository to prevent interactions between checkouts, compilation, and so on in the repository.

How to do it...

To begin bisecting, we simply type:

```
$ git bisect start
```

To mark the current commit (`HEAD -> bug_hunting`) as bad, we type:

```
$ git bisect bad
```

We also want to mark the last known good commit (`master`) as good:

```
$ git bisect good master
Bisecting: 11 revisions left to test after this (roughly 4
steps)
[9d2cd13d4574429dd0dcfeeb90c47a2d43a9b6ef] Build map part 11
```

This time, something happened. Git did a checkout of 9d2cd13, which it wants us to test and mark as good or bad. It also tells us there are 11 revisions to test after this and it can be done in approximately four steps. This is how the bisecting algorithm works; every time a commit is marked as good or bad, Git will checkout the middle one between the just marked one and the current one of opposite value. In this way, Git quickly narrows down the number of commits to check. It also knows that there are approximately four steps, and this makes sense since with 11 revisions left, the maximum number of tries is $\log_2(11) = 3.46$ before the faulty commit is found.

We can test with the `test.sh` script we created previously, and based on the return value, mark the commit as good or bad:

```
$ ./test.sh; test $? -eq 0 && git bisect good || git bisect
bad
# git bisect good
Bisecting: 5 revisions left to test after this (roughly 3
steps)
[c45cb51752a4fe41f52d40e0b2873350b95a9d7c] Build map part 16
```

The test marks the commit as good and Git checks out the next commit to be marked, until we hit the commit that introduces the bug:

```
$ ./test.sh; test $? -eq 0 && git bisect good || git bisect
bad
# git bisect bad
Bisecting: 2 revisions left to test after this (roughly 2
steps)
[83c22a39955ec10ac1a2a5e7e69fe7ca354129af] Bugs...
$ ./test.sh; test $? -eq 0 && git bisect good || git bisect
bad
# git bisect bad
Bisecting: 0 revisions left to test after this (roughly 1 step)
[670ab8c42a6cb1c730c7c4aa0cc26e5cc31c9254] Build map part 13
$ ./test.sh; test $? -eq 0 && git bisect good || git bisect
bad
```

```
# git bisect good
83c22a39955ec10ac1a2a5e7e69fe7ca354129af is the first bad commit
commit 83c22a39955ec10ac1a2a5e7e69fe7ca354129af
Author: HAL 9000 <aske.olsson@switch-gears.dk>
Date:   Tue May 13 09:53:45 2014 +0200
```

Bugs...

```
:100644 100644 8a13f6bd858aefb70ea0a7d8f601701339c28bb0
lafeaaa370a2e4656551a6d44053ee0ce5c3a237 M map.txt
```

After four steps, Git has identified the 1981eac1 commit as the first bad commit. We can end the bisect session and take a closer look at the commit:

```
$ git bisect reset
Previous HEAD position was 670ab8c... Build map part 13
Switched to branch 'bug_hunting'
Your branch is up-to-date with 'origin/bug_hunting'.
$ git show 83c22a39955ec10ac1a2a5e7e69fe7ca354129af
commit 83c22a39955ec10ac1a2a5e7e69fe7ca354129af
Author: HAL 9000 <aske.olsson@switch-gears.dk>
Date:   Tue May 13 09:53:45 2014 +0200
```

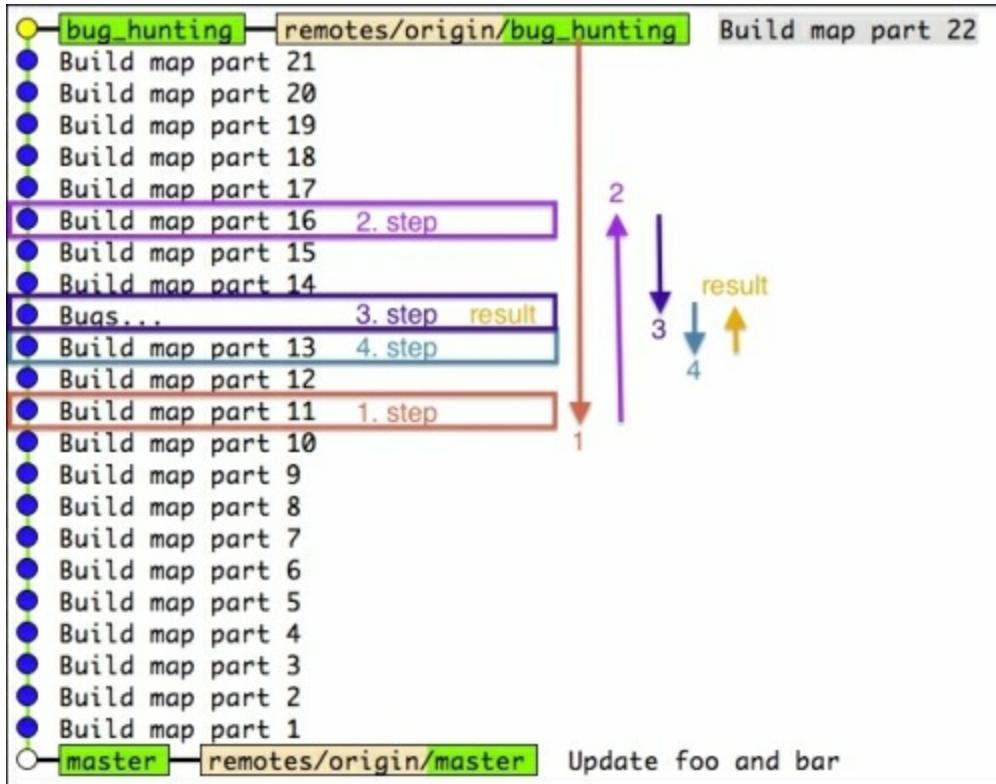
Bugs...

```
diff --git a/map.txt b/map.txt
index 8a13f6b..lafeaaa 100644
--- a/map.txt
+++ b/map.txt
@@ -34,6 +34,6 @@ Australia:
          .-./      |.      :    :,      \
          /       '-. /'      \_
-         .'                         *: Brisbane
-         .-'                         ;
-         |                         |
+         .'                         \_/
+         .-'                         (oo)
+         |                         //||\|
```

Clearly, there is a bug introduced with this commit.

The following annotated screenshot shows the steps taken by the bisect

session:



Note that the bisection algorithm actually hits the faulty commit in the third step, but it needs to look further to make sure that the commit isn't just a child commit of the faulty commit, and in fact is the commit that introduced the bug.

There's more...

Instead of running all the bisecting steps manually, it is possible to do it automatically by passing Git a script, makefile, or test to run on each commit. The script needs to exit with a **zero-status** to mark a commit as good and a **non-zero** status to mark it as bad. We can use the `test.sh` script we created at the beginning of this chapter for this. First, we set up the good and bad commits:

```
$ git bisect start HEAD master
Bisecting: 11 revisions left to test after this (roughly 4
steps)
[9d2cd13d4574429dd0dcfeeb90c47a2d43a9b6ef] Build map part 11
Then we tell Git to run the test.sh script and automatically
```

mark the commits:

```
$ git bisect run ./test.sh
running ./test.sh
Bisecting: 5 revisions left to test after this (roughly 3
steps)
[c45cb51752a4fe41f52d40e0b2873350b95a9d7c] Build map part 16
running ./test.sh
Bisecting: 2 revisions left to test after this (roughly 2
steps)
[83c22a39955ec10ac1a2a5e7e69fe7ca354129af] Bugs...
running ./test.sh
Bisecting: 0 revisions left to test after this (roughly 1 step)
[670ab8c42a6cb1c730c7c4aa0cc26e5cc31c9254] Build map part 13
running ./test.sh
83c22a39955ec10ac1a2a5e7e69fe7ca354129af is the first bad commit
commit 83c22a39955ec10ac1a2a5e7e69fe7ca354129af
Author: HAL 9000 <aske.olsson@switch-gears.dk>
Date:   Tue May 13 09:53:45 2014 +0200
```

Bugs...

```
:100644 100644 8a13f6bd858aefb70ea0a7d8f601701339c28bb0
lafeaaa370a2e4656551a6d44053ee0ce5c3a237 M  map.txt
biseect run success
```

Git found the same commit and we can now exit the bisecting session:

```
$ git bisect reset
Previous HEAD position was 670ab8c... Build map part 13
Switched to branch 'bug_hunting'
```

Using the blame command

The `bisect` command is good when you don't know where in your code there is a bug, but you can test for it and thereby find the commit that introduced it. If you already know where in the code the bug is but want to find the commit that introduced it, you can use `git blame`. The `blame` command will annotate every line in the file with the latest commit that touched that line, making it easy to find the commit ID and the full context of the commit.

Getting ready

We'll use the same repository and branch as in the `bisect` example:

```
$ git clone https://github.com/dvaske/cookbook-tips-tricks.git
$ cd cookbook-tips-tricks
$ git checkout bug_hunting
```

How to do it...

We know that the bug is in `map.txt` on lines 37-39. To annotate each line in the file with the commit ID and author, we'll run `git blame` on the file. We can further limit the search to specific lines with the `-L <from>, <to>` option:

```
$ git blame --date short -L 30,47 map.txt
828f25e7 (Dave Bowman 2014-05-13 30) Australia:
9d2cd13d (Frank Poole 2014-05-13 31)
9d2cd13d (Frank Poole 2014-05-13 32)
9d2cd13d (Frank Poole 2014-05-13 33)
fec0bd22 (Frank Poole 2014-05-13 34)
fec0bd22 (Frank Poole 2014-05-13 35)
fec0bd22 (Frank Poole 2014-05-13 36)
83c22a39 (HAL 9000 2014-05-13 37)
83c22a39 (HAL 9000 2014-05-13 38)
83c22a39 (HAL 9000 2014-05-13 39)
4c543b9c (Dave Bowman 2014-05-13 40)
4c543b9c (Dave Bowman 2014-05-13 41)
4c543b9c (Dave Bowman 2014-05-13 42)
7ad3da42 (Heywood R. Floyd 2014-05-13 43)
7ad3da42 (Heywood R. Floyd 2014-05-13 44)
7ad3da42 (Heywood R. Floyd 2014-05-13 45)
c45cb517 (Heywood R. Floyd 2014-05-13 46)
c45cb517 (Heywood R. Floyd 2014-05-13 47)
```

The blame output shows the history of changes to `map.txt` from line 30 to 47. The commits are listed with their authors, dates, and the location they were committed from. The diagram illustrates the flow of changes from Perth to Darwin, then to Melbourne, and finally to Brisbane.

From the output, it can be clearly seen that the commit with the ID 83c22a39 by HAL 9000 introduced the bug.

There's more...

The `blame` command can be used even if the file has been refactored and the code has been moved around. With the `-M` option, the `blame` command can detect lines that have been moved around in the file and with the `-C` option, Git can detect lines that were moved or copied from other files in the same commit. If the `-C` option is used three times `-CCC`, the `blame` command will find lines that were copied from other files in any commit.

Color UI in the prompt

By default, Git has no colors when displaying information in the terminal. However, displaying colors is a feature of Git that is only a configuration away.

Getting ready

We'll use the `cookbook-tips-tricks` repository:

```
$ git clone https://github.com/dvaske/cookbook-tips-tricks.git  
$ cd cookbook-tips-tricks
```

How to do it...

First, we'll edit and add `foo`:

```
$ echo "And another line" >> foo  
$ git add foo
```

Change `foo` some more, but don't add it to the staging area:

```
$ echo "Last line ...so far" >> foo
```

Create a new file called `test`:

```
$ touch test
```

The `git status` command will show us the status:

```
$ 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:   foo
```

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: foo
```

```
Untracked files:
```

```
(use "git add <file>..." to include in what will be  
committed)
```

```
test
```

We can set the `color.ui` configuration to `auto` or `true` to get color in the UI when required:

```
$ git config --global color.ui true  
$ 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: foo
```

```
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: foo
```

```
Untracked files:
```

```
(use "git add <file>..." to include in what will be  
committed)
```

```
test
```

There's more...

The `color.ui` configuration works with a long range of Git commands `diff`, `log`, and `branch` included. The following is an example of `git log` when setting `color.ui` to `true`:

```
$ git log --oneline --decorate --graph  
* c111003 (HEAD, origin/master, origin/HEAD, master) Update foo
```

and bar
* 270e97b Add bar
* 43fd490 Initial commit, adds foo

Autocompletion

Git comes with built-in support for auto-completion of Git commands for the `bash` and `zsh` shells. So if you use either of these shells, you can enable the auto-completion feature and let the `<tab>` option help you complete commands.

Getting ready

Generally, the auto-completion feature is distributed with the Git installation, but it is not enabled by default on all platforms or distributions. To enable it, we need to find the `git-completion.bash` file distributed/installed with the Git installation.

Linux

For Linux users, the location may vary depending on the distribution. Generally, the file can be found at `/etc/bash_completion.d/git-completion.bash`.

Mac

For mac users, it can generally be found at
`/Library/Developer/CommandLineTools/usr/share/git-core/git-completion.bash`.

If you installed Git from Homebrew, it can be found at
`/usr/local/Cellar/git/1.9.1/etc/bash_completion.d/git-completion.bash`.

Windows

With the **Msysgit** installation on Windows, the completion functions are already enabled in the Git bash shell it bundles.

If you can't find the file on your system, you can grab the latest version from <https://github.com/git/git/blob/master/contrib/completion/git-completion.bash> and install it in your home directory.

How to do it...

To enable the completion feature, you need to run the `source` command on the completion file, which you can do by adding the following lines to your `.bashrc` or `.zshrc` file depending on your shell and the location of the Git completion file:

```
if [ -f /etc/bash_completion.d/git-completion.bash ]; then
    source /etc/bash_completion.d/git-completion.bash
fi
```

How it works...

Now, you are ready to try. Switch to an existing Git repository, for example, `cookbook-tips-tricks` and type the following commands:

```
$ git che<tab><tab>
checkout      cherry      cherry-pick
```

You can add another `c<tab>` and the command will autocomplete to `checkout`. But the autocompletion feature doesn't only complete commands, it can also help you complete branch names, and so on, so you can continue with the `checkout` and write `mas<tab>`. You should be able to see the output completed to the `master` branch unless you are in a repository where there are several branches starting with `mas`.

There's more...

The completion feature also works with options; this is quite useful if you can't remember the exact option but you may remember some of it, for example, when using `git branch`:

```
git branch --<tab><tab>
--abbrev=          --merged          --set-upstream-to=
--color           --no-abbrev       --track
--contains        --no-color        --unset-upstream
--edit-description --no-merged      --verbose
--list            --no-track
```

Bash prompt with status information

Another cool feature Git provides is to have the prompt display status information if the current working directory is a Git repository.

Getting ready

For the status information prompt to work, we also need to source another file, `git-prompt.sh`, which is usually distributed with the Git installation and located in the same directory as the completion file.

How to do it...

In your `.bashrc` or `.zshrc` file, add the following code snippet, again depending on your shell and the location of the `git-prompt.sh` file:

```
if [ -f /etc/bash_completion.d/git-prompt.sh ]; then
    source /etc/bash_completion.d/git-prompt.sh
fi
```

How it works...

To make use of the command prompt, we must change the `PS1` variable; usually this is set to something like this:

```
PS1=' \u@\h: \w\$ '
```

The preceding command shows the current user, an @ sign, the host name, the current working directory relative to the user's home directory, and finally a \$ character:

```
aske@yggdrasil:~/cookbook-tips-tricks$
```

We can change this to add a branch name after the working directory by adding `$(_git_ps1 " (%s)")` to the `PS1` variable:

```
PS1=' \u@\h: \w\$(_git_ps1 " (%s)") \$ '
```

Our prompt will now look like this:

```
aske@yggdrasil:~/cookbook-tips-tricks (master) $
```

It is also possible to show the state of the working tree, index, and so on. We can enable these features by exporting some environment variables in the `.bashrc` file that `git-prompt.sh` picks up.

The following environment variables can be set:

Variable	Value	Effect
<code>GIT_PS1_SHOWDIRTYSTATE</code>	Nonempty	Shows * for unstaged changes and + for staged changes
<code>GIT_PS1_SHOWSTASHSTATE</code>	Nonempty	Shows a \$ character if something is stashed
<code>GIT_PS1_SHOWUNTRACKEDFILES</code>	Nonempty	Shows a % character if there are untracked files in the repository
<code>GIT_PS1_SHOWUPSTREAM</code>	auto verbose name Legacy Git svn	Auto shows whether you are behind (<) or ahead (>) of the upstream branch. A <> value is displayed if the branch is diverged and = if it is up to date. Verbose shows the number of commits behind/ahead. Name shows the upstream name. Legacy is verbose for old versions of Git. Git compares <code>HEAD</code> to <code>@{upstream}</code> . SVN compares <code>HEAD</code> to <code>svn upstream</code> .
<code>GIT_PS1_DESCRIBE_STYLE</code>	contains branch describe default	Displays extra information when on a detached <code>HEAD</code> . Contains is relative to a newer annotated tag (v1.6.3.2~35). Branch is relative to a newer tag or branch (master~4). Describe is relative to an older annotated tag (v1.6.3.1-13-gdd42c2f). Default is the tag that matches exactly.

Let's try to set some of the variables in the `~/.bashrc` file:

```
export GIT_PS1_SHOWUPSTREAM=auto
export GIT_PS1_SHOWDIRTYSTATE=enabled
PS1='\u@\h:\w$(_git_ps1 "%s") \$ '
```

Let us see the `~/.bashrc` file in action:

```
aske@yggdrasil:~ $ cd cookbook-tips-tricks/
aske@yggdrasil:~/cookbook-tips-tricks (master=) $ touch test
aske@yggdrasil:~/cookbook-tips-tricks (master=) $ git add test
aske@yggdrasil:~/cookbook-tips-tricks (master +=) $ echo
"Testing" > test
aske@yggdrasil:~/cookbook-tips-tricks (master *+=) $ git commit
-m "test"
[master 5c66d65] test
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 test
aske@yggdrasil:~/cookbook-tips-tricks (master *>) $
```

When using the `__git_ps1` option, Git will also display information when merging, rebasing, bisecting, and so on. This is very useful and a lot of `git status` commands suddenly become unnecessary as you have the information right there in the prompt.

There's more...

What is a terminal these days without some colors? The `git-prompt.sh` script also supports this. All we need to do is set the `GIT_PS1_SHOWCOLORHINTS` variable to a nonempty value and instead of using `PS1`, we need to use `PROMPT_COMMAND`. Let's change `~/.bashrc`:

```
export GIT_PS1_SHOWUPSTREAM=auto
export GIT_PS1_SHOWDIRTYSTATE=enabled
export GIT_PS1_SHOWCOLORHINTS=enabled
PROMPT_COMMAND='__git_ps1 "\u001b[0;32m\u001b[0m\h:\u001b[0;34m\u001b[0m\w" "\u001b[0;31m\u001b[0m\$ "'
```

If we redo the same scenario as the previous one, we get the following:

```
aske@yggdrasil:~ $ cd cookbook-tips-tricks/
aske@yggdrasil:~/cookbook-tips-tricks (master=) $ touch test
aske@yggdrasil:~/cookbook-tips-tricks (master=) $ git add test
aske@yggdrasil:~/cookbook-tips-tricks (master +=) $ echo
"Testing" > test
aske@yggdrasil:~/cookbook-tips-tricks (master *+=) $ git commit
-m "test"
[master 0cb59ca] test
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 test
```

```
aske@yggdrasil:~/cookbook-tips-tricks (master *->) $
```

See also

If you are using zsh or just want to try something new with many features, such as completion, Git support, and so on, you should take a look at the oh-my-zsh framework available for zsh at <https://github.com/robbyrussell/oh-my-zsh>.

More aliases

In [Chapter 2, Configuration](#), we saw how we can create aliases and a few examples of them. In this example, we will see some more examples of the useful aliases.

Getting ready

Clone the `cookbook-tips-tricks` repository and checkout the `aliases` branch:

```
$ git clone https://github.com/dvaske/cookbook-tips-tricks.git
$ cd cookbook-tips-tricks
$ git checkout aliases
```

How to do it...

Here, we'll see some examples of aliases with a short description of each of them and an example of how to use them. The aliases are just made for the local repository; use `--global` to make them available for all the repositories.

- Show the current branch only:

```
$ git config alias.b "rev-parse --abbrev-ref HEAD"
$ git b
aliases
```

- Show a compact graph history view with colors:

```
git config alias.graph "log --graph --
pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s
%Cgreen(%cr) %C(bold blue)<%an>%Creset' --abbrev-commit --
date=relative"
git graph origin/conflict aliases
```

```
* a43eaa9 - (HEAD, origin/aliases, aliases) Better spaceship design (58 minutes ago) <Aske Olsson>
| * 918fe4e - (origin/conflict) Spaceship upgrade (60 minutes ago) <Aske Olsson>
|/
* 8fc1819 - Adds spaceship (64 minutes ago) <Aske Olsson>
* c73d2ef - Adds directory structure (67 minutes ago) <Aske Olsson>
* c111003 - (origin/master, origin/HEAD, master) Update foo and bar (3 days ago) <Aske Olsson>
* 270e97b - Add bar (3 days ago) <Aske Olsson>
* 43fd490 - Initial commit, adds foo (3 days ago) <Aske Olsson>
```

- When resolving a conflicted merge, get a list of the conflicted/unmerged files:

```
$ git config alias.unmerged '!git ls-files --unmerged | cut -f2 | sort -u'
```

We can see the previous command in action by merging the origin/conflict branch:

```
$ git merge origin/conflict
Auto-merging spaceship.txt
CONFLICT (content): Merge conflict in spaceship.txt
Automatic merge failed; fix conflicts and then commit the result.
```

Check the output of `git status` first:

```
$ git status
On branch aliases
Your branch is up-to-date with 'origin/aliases'.

You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

    both modified:      spaceship.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

Let's see what the unmerged alias does:

```
$ git unmerged
spaceship.txt
```

Abort the merge:

```
$ git merge --abort
```

- Shorthand status as follows:

```
git config alias.st "status"
```

```
git st
On branch aliases
Your branch is up-to-date with 'origin/aliases'.
```

```
nothing to commit, working directory clean
```

- A shorter status with branch and file information:

```
$ git config alias.s 'status -sb'
```

```
Modify foo and create an untracked file test:
```

```
$ touch test
$ echo testing >> foo
```

```
Try the s alias:
```

```
$ git s
## aliases...origin/aliases
 M foo
?? test
```

- Show the latest commit with some stats:

```
$ git config alias.ll "log -1 --shortstat"
$ git ll
commit a43eaa9b461e811eeb0f18cce67e4465888da333
Author: Aske Olsson <aske.olsson@switch-gears.dk>
Date:   Wed May 14 22:46:32 2014 +0200
```

```
Better spaceship design
```

```
1 file changed, 9 insertions(+), 9 deletions(-)
```

- This gives the same view as the previous but for the five latest commits (the output is not shown):

```
$ git config alias.l5 "log -5 --decorate --shortstat"
```

- A commit listing with statistics on the changed files in colors can be displayed using the following command:

```
$ git config alias.ll "log --
pretty=format:\"%C(yellow)%h%Creset %s %Cgreen(%cr) %C(bold
blue)<%an>%Creset %Cred%d%Creset\" --numstat"
$ git ll -5
```

```

a43eaa9 Better spaceship design (58 minutes ago) <Aske Olsson> (HEAD, origin/aliases, aliases)
9         9      spaceship.txt

8fc1819 Adds spaceship (64 minutes ago) <Aske Olsson>
43        0      spaceship.txt

c73d2ef Adds directory structure (67 minutes ago) <Aske Olsson>
1         0      sub/directory/example/readme.txt

c111003 Update foo and bar (3 days ago) <Aske Olsson> (origin/master, origin/HEAD, master)
7         0      bar
7         0      foo

270e97b Add bar (3 days ago) <Aske Olsson>
1         0      bar

```

- Show the upstream/tracking branch:

```

$ git config alias.upstream "rev-parse --symbolic-full-name
--abbrev-ref=strict HEAD@{u}"
$ git upstream
origin/aliases

```

- Show details of ID/SHA-1 (commit, tag, tree, blob):

```

git config alias.details "cat-file -p"
git details HEAD
tree bdfdaacbb29934b239db814e599342159c4390dd
parent 8fc1819f157f2c3c25eb973c2a2a412ef3d5517a
author Aske Olsson <aske.olsson@switch-gears.dk> 1400100392
+0200
committer Aske Olsson <aske.olsson@switch-gears.dk>
1400100392 +0200

```

Better spaceship design

- Show the numbers of "cd-ups", . . . /, needed to go to the repository root using following command:

```

$ git config alias.root "rev-parse --show-cdup"
$ cd sub/directory/example
$ pwd
/path/to/cookbook-tips-tricks/sub/directory/example
$ git root
../../...
$ cd $(git root)
$ pwd
/path/to/cookbook-tips-tricks

```

- The path of the repository on the filesystem:

```
$ git config alias.path "rev-parse --show-toplevel"  
$ git path  
/path/to/cookbook-tips-tricks
```

- Abandon whatever changes we have in the index, working tree, and possibly also the commits and reset the working tree to a known state (commit ID). Do not touch the untracked files. We need a ref as an argument for the state of the repository to be restored, for example, HEAD:

```
$ git config alias.abandon "reset --hard"  
$ echo "new stuff" >> foo  
$ git add foo  
$ echo "other stuff" >> bar  
$ git s  
## aliases...origin/aliases  
 M bar  
M foo  
?? test  
$ git abandon HEAD  
$ git s  
## aliases...origin/aliases  
?? test
```

Interactive add

The exposed staging area Git offers sometimes lead to confusion, especially when adding a file, changing it a bit, and then adding the file again to be able to commit the changes made after the first add. While this can seem a bit cumbersome to add the file after every little change, it is also a big advantage that you can stage and unstage changes. With the `git add` command, it is even possible to only add some changes to a file in the staging area. This comes in handy especially if you make a lot of changes to a file and for example, want to split the changes into bug fixes, refactoring, and features. This example will show how you can easily do this.

Getting ready

Again, we'll use the `cookbook-tips-tricks` repository. Clone it and check out the interactive branch:

```
$ git clone https://github.com/dvaske/cookbook-tips-tricks.git
$ cd cookbook-tips-tricks
$ git checkout interactive
```

How to do it...

First, we need some changes to be added; we do this by a resetting the latest commit:

```
$ git reset 'HEAD^'
Unstaged changes after reset:
M    liberty.txt
```

Now, we have a modified file. To start the interactive add, we can either run the `git add -i` or `git add -p filename`. The `-i` option brings up an interface where all the different files in the modified state can be added interactively one at a time. The `add -p/--patch` option is simpler and just gives you the option to add parts of the file specified:

```
$ git add -p liberty.txt
diff --git a/liberty.txt b/liberty.txt
```

```
index 8350a2c..9638930 100644
--- a/liberty.txt
+++ b/liberty.txt
@@ -8,6 +8,13 @@
    WW) ,WWW)
    7W) ,WWWW'
    'WWWWWW'
+
    9---W)
+
    , , --WPL=YXW===
+
    (P) ,CY: ,I/X'F9P
+
    WUT===== /====9)
+
    -HP+----Y (C=9W)
+
    '9Y3'-'-OWPT-
+
    'WWLUIECW
    (:7L7C7'
    ,P--=YWFL
    Y-=:9)UW:L
Stage this hunk [y,n,q,a,d/,j,J,g,e,?]?
```

Git asks you whether you want to stage the previous change (the hunk), but also shows quite a lot of options, which can expand a little bit if you type ?:

```
Stage this hunk [y,n,q,a,d/,j,J,g,e,?]?
y - stage this hunk
n - do not stage this hunk
q - quit; do not stage this hunk nor any of the remaining ones
a - stage this hunk and all later hunks in the file
d - do not stage this hunk nor any of the later hunks in the
file
g - select a hunk to go to
/ - search for a hunk matching the given regex
j - leave this hunk undecided, see next undecided hunk
J - leave this hunk undecided, see next hunk
k - leave this hunk undecided, see previous undecided hunk
K - leave this hunk undecided, see previous hunk
s - split the current hunk into smaller hunks
e - manually edit the current hunk
? - print help
```

There are a lot of options, but with the help text, they are quite self-explanatory. Let's add the current hunk, `y`, and look at the next one:

```
Stage this hunk [y,n,q,a,d/,j,J,g,e,?]? y
@@ -17,16 +24,17 @@
```

```

7WYW) ) PW W
7WH) ) ,WC)
7L--/XY)
+DEBUG: Don't include this line...
    9+-,KY7)
    W9-Y3+7)
    W'=9WI7)
    ,W '-YY)
-
    W   ::W
-
    ,T   :X)
-
    ()   '9W  'L.           ,-
(C   =:9   '9L           ,T
-
    ()   ,,-7)   7WL           ,F'
-
    ()   , T9)   '9WL           ,YF
+
    W   ::W           ,
+
    ,T   :X)           ()
+
    ()   '9W  'L.           ()           ,-
+
    (C   =:9   '9L           ()           ,T
+
    ()   ,,-7)   7WL   WW           ,F'
+
    ()   , T9)   '9WL   --           ,YF
    ()   '-/(W   -==+PE9P7==O)           -,,
    'W, ,   T+/WX=L-. ,WP+() +3L3,) ,=WL   ----T-
    7)   -,YW '=-9WPL+PT-- ':-L/=9WP=-'

```

Stage this hunk [y,n,q,a,d,/,,K,j,J,g,s,e,?]?

Whoa! It looks like there is a debug line there that we should not add. Let's split the hunk, s:

```

Stage this hunk [y,n,q,a,d,/,,K,j,J,g,s,e,?]? s
Split into 2 hunks.
@@ -17,7 +24,8 @@
    7WYW) ) PW W
    7WH) ) ,WC)
    7L--/XY)
+DEBUG: Don't include this line...
    9+-,KY7)
    W9-Y3+7)
    W'=9WI7)
    ,W '-YY)

```

Stage this hunk [y,n,q,a,d,/,,K,j,J,g,e,?]?

We don't want the output to be staged, use n, but we'll stage the next hunk and all the hunks later in the a file:

```
Stage this hunk [y,n,q,a,d,/,,K,j,J,g,e,?]? n
```

```

@@ -20,13 +28,13 @@
      9+-,KY7)
      W9-Y3+7)
      W'=9WI7)
      ,W  '-YY)
-
-      W    ::W
-      ,T    :X)
-      ()    '9W  'L.
-      (C    =:9  '9L
-      ()    ,,-7)    7WL
-      ()    , T9)    '9WL
+      W    ::W
+      ,T    :X)
+      ()    '9W  'L.
+      (C    =:9  '9L
+      ()    ,,-7)    7WL
+      ()    , T9)    '9WL
      ()    '-/(W    ---+PE9P7==O)
      'W, ,  T+/WX=L-. ,WP+() +3L3,) ,=WL  ----T-
      7)    -,YW  '--=9WPL+PT-- ':--L/=9WP=-

```

Stage this hunk [y,n,q,a,d,/ ,K,j,J,g,e,?] ? a

Let's see the current status and difference between the working directory and staging area:

```

$ git status
On branch interactive
Your branch is behind 'origin/interactive' by 1 commit, and can
be fast-forwarded.
(use "git pull" to update your local branch)

```

Changes to be committed:
 (use "git reset HEAD <file>..." to unstage)

modified: liberty.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: liberty.txt

```

$ git diff
diff --git a/liberty.txt b/liberty.txt
index 035083e..9638930 100644
--- a/liberty.txt

```

```

+++ b/liberty.txt
@@ -24,6 +24,7 @@
    7WYW) ) PW W
    7WH) ),WC)
    7L--/XY)
+DEBUG: Don't include this line...          9+- ,KY7)
    W9-Y3+7)
    W'=9WI7)

```

Perfect! We got all the changes staged except the debug line, so the result can be committed:

```
$ git commit -m 'Statue of liberty completed'
[interactive 1ccb885] Statue of liberty completed
 1 file changed, 36 insertions(+), 29 deletions(-)
```

There's more...

As mentioned earlier, it is also possible to use `git add -i` to interactively add files. If we do this after resetting our branch, we would get the following menu:

```
$ git add -i
      staged      unstaged path
 1:   unchanged      +37/-29 liberty.txt

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

The eight options pretty much do what they say. We can choose the patch option to get into the patch menu as we saw previously, but first we have to choose which file to add patches for:

```
What now> p
      staged      unstaged path
 1:   unchanged      +37/-29 liberty.txt
Patch update>> 1
      staged      unstaged path
* 1:   unchanged      +37/-29 liberty.txt
Patch update>>
diff --git a/liberty.txt b/liberty.txt
index 8350a2c..9638930 100644
```

```
--- a/liberty.txt
+++ b/liberty.txt
...
```

Once we have chosen the files, we want to add patches so they get a * character in the menu. To begin the patching, just click on <return>. When you're done, you'll return to the menu and can quit, review, revert, and so on.

Interactive add with Git GUI

The interactive features of `git add` are really powerful in order to create clean commits that only contain a single logical change even though it was coded as a mix of feature adding and bug fixing. The downside of the interactive `git add` feature is that it is hard to get an overview of all the changes that exist in the file when only being showed one hunk at a time. To get a better overview of the changes and still be able to only add selected hunks (and even single lines), we can use `git gui`. Git GUI is normally distributed with the Git installation (MsysGit on Windows) and can be launched from the command line: `git gui`. If your distribution doesn't have Git GUI available, you can probably install it from the package manager named `git-gui`.

Getting ready

We'll use the same repository as in the last example and reset it to the same state so we can perform the same adds with Git GUI:

```
$ git clone https://github.com/dvaske/cookbook-tips-tricks.git
$ cd cookbook-tips-tricks
$ git checkout interactive
$ git reset HEAD^
```

How to do it...

Load up Git GUI in the `cookbook-tips-tricks` repository. Here, you can see the unstaged changes (files) in the top-left side and the staged changes (files) under it. The main window will display the unstaged changes in the current marked file. You can right-click on a hunk and see a context menu with options for staging and so on. The first hunk shown by Git GUI is much larger than what we saw before with `git add -p`. Choose **Show Less Context** to split the hunk, as shown in the following screenshot:

Current Branch: interactive

Unstaged Changes

liberty.txt

Modified, not staged

File: liberty.txt

```
@@ -6,29 +6,37 @@  
    YP ,W'  
    ,W) ,WW.'  
    WW) ,WWW)  
    ?W),WWWW'  
    `WWWWWW'  
+    9---W)  
+    ,--WPL=YXW---  
+    (P),CY:,I/X'F9P  
+    WUT=====/-==9)  
+    -HP+----Y(CC=9W)  
+    '9Y3'-'-OWPT-  
    'WWLUIECW  
    (:?L7C7'  
    ,P---YWFL  
    Y--:9)UW:L  
    3-'9=WU/.7  
    ,WP9HTFUW'()  
    9W7W))UF 9)  
    7WYW))PW W  
    7WH)),WC)  
    7L--/XY)  
+DEBUG: Don't include this line.  
    9+-,KY7)  
    W9-Y3+7)  
    W'=9WI7)  
    ,W '-YY)  
-    W ::W
```

Stage Hunk For Commit
Stage Line For Commit
Show Less Context
Show More Context
Refresh
Copy
Select All
Copy All
Decrease Font Size
Increase Font Size
Encoding ►
Options...

Commit Message: New Commit Amend Last Commit

Rescan
Stage Changed
Sign Off
Commit
Push

Ready.

Now, we get a smaller hunk like before, as shown in the following screenshot:

Current Branch: interactive

Unstaged Changes

liberty.txt

Modified, not staged

```
@@ -7,8 +7,15 @@
 ,W) ,WW.
 WW) ,WWW)
 7W),WWWW'
 `WWWWWW'
+
 9---W)
+ ,--WPL=YXW===
+(P),CY:,I/X'F9P
+WUT===== /==9)
+ -HP+----Y(C=9W)
+ '9Y3'-'-OWPT-
+ 'WWLUIECW
 (:7L7C7'
 ,P---YWFL
 Y-=:9)UW:L
 3-'9=WU/.7
@@ -16,18 +23,19 @@
 9W7W))UF 9)
 7WYW))PW W
```

File: liberty.txt

Stage Hunk For Commit

Stage Line For Commit

Show Less Context

Show More Context

Refresh

Copy

Select All

Copy All

Decrease Font Size

Increase Font Size

The first hunk we just choose to add, **Stage Hunk For Commit**, and the next hunk moves to the top of the screen:

Current Branch: interactive

Unstaged Changes

liberty.txt

Modified, not staged

```
@@ -16,18 +23,19 @@
 9W7W))UF 9)
 7WYW))PW W
 7WH)),WC)
 7L--/XY)
+DEBUG: Don't include this line...
 9+-,KY7)
 W9-Y3+7)
 W'=9WI7)
 ,W '-YY)
-
 W ::W
-,T :X)
-(O '9W 'L.
-(C =:9 '9L
-(O ,,-7) 7WL
-(O , T9) '9WL ,YF
+
 W ::W
+,T :X) O
+(O '9W 'L. O
+(C =:9 '9L O
+(O ,,-7) 7WL WW
+(O , T9) '9WL -- ,YF
(O '-/(W ---+PE9P7==0)
 'W, , T+/WX=L-. ,WP+C)+3L3,,=WL
 ?) -,YW '--=9WPL+PT-- ':--L/=9W
 'W-,--,++W. WWHP ,,-/ .9CP:
@@ -50,24 +58,24 @@
```

File: liberty.txt

Stage Hunk For Commit

Stage Lines For Commit

Show Less Context

Show More Context

Refresh

Copy

Select All

Copy All

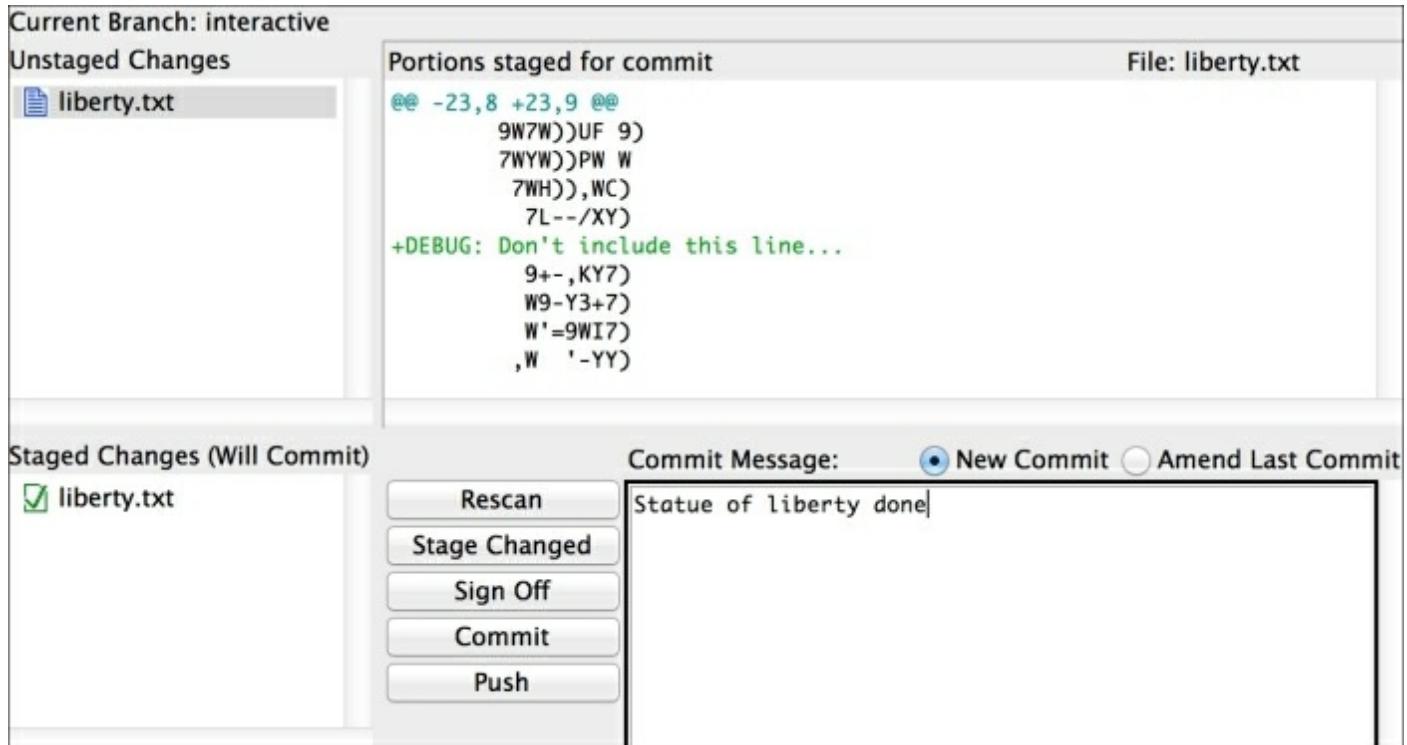
Decrease Font Size

Increase Font Size

Encoding ►

Here, we can select the lines we want to add, instead of performing

another split, and stage those lines: **Stage Lines For Commit**. We can add the rest of the hunks except the one with the debug line. Now, we are ready to create a commit and we can do so from the Git GUI. We can just write the commit message in the field at the bottom of the screen and hit **Commit**:



Ignoring files

For every repository, there are usually certain types of files you don't want tracked in the repository. The files can be configuration files, build output, or just backup files created by the editor when editing the file. To avoid these files showing up in the untracked files section of the `git status` output, it is possible to add them to a file called `.gitignore`. Entries in this file that match files in the working directory will not be considered by `git status`.

Getting ready

Clone the `cookbook-tips-tricks` repository and check out the `ignore` branch:

```
$ git clone https://github.com/dvaske/cookbook-tips-tricks.git
$ cd cookbook-tips-tricks
$ git checkout ignore
```

How to do it...

First, we'll create some files and directories:

```
$ echo "Testing" > test.txt
$ echo "Testing" > test.txt.bak
$ mkdir bin
$ touch bin/foobar
$ touch bin/frotz
```

Let's see the output of `git status`:

```
$ git status
On branch ignore
Your branch is up-to-date with 'origin/ignore'.
```

Untracked files:

```
(use "git add <file>..." to include in what will be
committed)
```

```
test.txt
```

```
nothing added to commit but untracked files present (use "git add" to track)
```

Only the `test.txt` file showed up in the output. This is because the rest of the files are ignored by Git. We can check the content of `.gitignore` to see how this happened:

```
cat .gitignore
*.config
*.bak
```

```
# Java files
*.class
```

```
bin/
```

This means that `*.bak`, `*.class`, `*.config`, and everything in the `bin` directory are being ignored by Git.

If we try to add files in a path ignored by Git, for example `bin`, it will complain:

```
$ git add bin/frotz
The following paths are ignored by one of your .gitignore
files:
bin/frotz
Use -f if you really want to add them.
```

But, it also tells us an option to use if we really want to add it, `-f`:

```
$ git add -f bin/frotz
$ git status
On branch ignore
Your branch is up-to-date with 'origin/ignore'.
```

```
Changes to be committed:
(use "git reset HEAD <file>..." to unstage)
```

```
  new file:   bin/frotz
```

```
Untracked files:
(use "git add <file>..." to include in what will be
committed)
```

test.txt

If we ignore the `foo` file, which is already tracked, and modify it, it still shows up in the status since tracked files are not ignored:

```
$ echo "foo" >> .gitignore
$ echo "more testing" >> foo
$ git status
On branch ignore
Your branch is up-to-date with 'origin/ignore'.

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   bin/frotz

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:   .gitignore
    modified:   foo

Untracked files:
  (use "git add <file>..." to include in what will be
committed)

  test.txt
```

Let's add and commit `foo`, `.gitignore` and the content of the current staging area:

```
$ git add foo .gitignore
$ git commit -m 'Add bin/frotz with force, foo & .gitignore'
[ignore fc60b44] Add bin/frotz with force, foo & .gitignore
 3 files changed, 2 insertions(+)
 create mode 100644 bin/frotz
```

There's more...

It is also possible to ignore files of a repository without the `.gitignore` files. You can put your ignored files in a global ignore file, for example `~/.gitignore_global`, and globally configure Git to also consider entries

in this file to be ignored:

```
$ git config --global core.excludesfile ~/.gitignore_global
```

You can also do it per repository in the `.git/info/exclude` file. If you use either of these options, you won't be able to easily share the ignored file; they can't be added to the repository as they are stored outside it. Sharing the `.gitignore` files is much easier; you just add and commit it to Git. But, let's see how the other options work:

```
$ echo "*.test" > .git/info/exclude
$ touch test.test
$ git status
On branch ignore
Your branch is ahead of 'origin/ignore' by 1 commit.
  (use "git push" to publish your local commits)
```

Untracked files:

```
(use "git add <file>..." to include in what will be
committed)
```

```
test.txt
```

```
nothing added to commit but untracked files present (use "git
add" to track)
```

```
$ ls
bar      bin      foo
test.test  test.txt  test.txt.bak
```

We can see that the `.test` file didn't show up in the `status` output and that the ignored files exists in the working directory.

See also...

There is a wide range of files ignored commonly, for example, to avoid accidentally adding text editor backup files, `*.swp`, `*~`. and `*.bak` are commonly ignored. If you are working on a Java project, you might add `*.class`, `*.jar`, `*.war` to your `.gitignore` and `*.o`, `*.elf`, `*.lib` if you are working on a C project. Github has a repository dedicated to collect Git ignore files for different programming languages and editors/IDEs. You can find it at <https://github.com/github/gitignore>.

Showing and cleaning ignored files

Ignoring files is useful to filter noise from the output of `git status`. But, sometimes it is required to check which files are ignored. This example will show you how to do that.

Getting ready

We'll continue in the repository from the last example.

How to do it...

To show the files we have ignored, we can use the `clean` command. Normally, the `clean` command will remove the untracked files from the working directory but it is possible to run this in a dry-run mode, `-n`, where it just shows what will happen.

```
$ git clean -Xnd
Would remove bin/foobar
Would remove test.test
Would remove test.txt.bak
```

The options used in the preceding command specify the following:

- `-n`, `--dry-run`: Only lists that will be removed
- `-X`: Removes only the files ignored by Git
- `-d`: Removes the untracked directories in addition to the untracked files

The ignored files can also be listed with the `ls-files` command:

```
$ git ls-files -o -i --exclude-standard
bin/foobar
test.test
test.txt.bak
```

Where the option `-o`, `--others` shows the untracked files, the option `-i`,

--ignored shows only the ignored files, and --exclude-standard, use the standard exclusion files `.git/info/exclude` and `.gitignore` in each directory, and the user's global exclusion file.

There's more...

If we need to remove the ignored files, we can of course use `git clean` to do this; instead of the dry-run option, we pass the force option, `-f`:

```
$ git clean -Xfd
Removing bin/foobar
Removing test.test
Removing test.txt.bak
```

To remove all the untracked files and not just the ignored files, use `git clean -xfd`. The lower case `x` means we don't use the ignore rules, we just remove everything that is not tracked by Git.

Part III. Module 3

Mastering Git

Attain expert-level proficiency with Git for enhanced productivity and efficient collaboration by mastering advanced distributed version control features

Chapter 1. Git Basics in Practice

This book is intended for novice and advanced Git users to help them on their road to mastering Git. Therefore the following chapters will assume that the reader knows the basics of Git, and has advanced past the beginner stage.

This chapter will serve as a reminder of version control basics with Git. The focus will be on providing practical aspects of the technology, showing and explaining basic version control operations in the example of the development of a sample project, and collaboration between two developers.

In this chapter we will recall:

- Setting up a Git environment and Git repository (`init`, `clone`)
- Adding files, checking status, creating commits, and examining the history
- Interacting with other Git repositories (`pull`, `push`)
- How to resolve a merge conflict
- Creating and listing branches, switching to a branch, and merging
- How to create a tag

An introduction to version control and Git

A **version control system** (sometimes called **revision control**) is a tool that lets you track the history and attribution of your project files over time (stored in a **repository**), and which helps the developers in the team to work together. Modern version control systems help them work simultaneously, in a non-blocking way, by giving each developer his or her own sandbox, preventing their work in progress from conflicting, and all the while providing a mechanism to merge changes and synchronize work.

Distributed version control systems such as Git give each developer his or her own copy of the project's history, a **clone** of a repository. This is what makes Git fast: nearly all operations are performed locally, and are flexible: you can set up repositories in many ways. Repositories meant for developing also provide a separate **working area** (or a **worktree**) with project files for each developer. The branching model used by Git enables cheap local branching and flexible branch publishing, allowing to use branches for context switching and for sandboxing different works in progress (making possible, among other things, a very flexible **topic branch** workflow).

The fact that the whole history is accessible allows for long-term undo, rewinding back to last working version, and so on. Tracking ownership of changes automatically makes it possible to find out who was responsible for any given area of code, and when each change was done. You can compare different revisions, go back to the revision a user is sending a bug report against, and even automatically find out which revision introduced a regression bug. The fact that Git is tracking changes to the tips of branches with **reflog** allows for easy undo and recovery.

A unique feature of Git is that it enables explicit access to the staging area for creating commits (new revisions of a project). This brings additional flexibility to managing your working area and deciding on the shape of a future commit.

All this flexibility and power comes at a cost. It is not easy to master using Git, even though it is quite easy to learn its basic use. This book will help you attain this expertise, but let's start with a reminder about basics with Git.

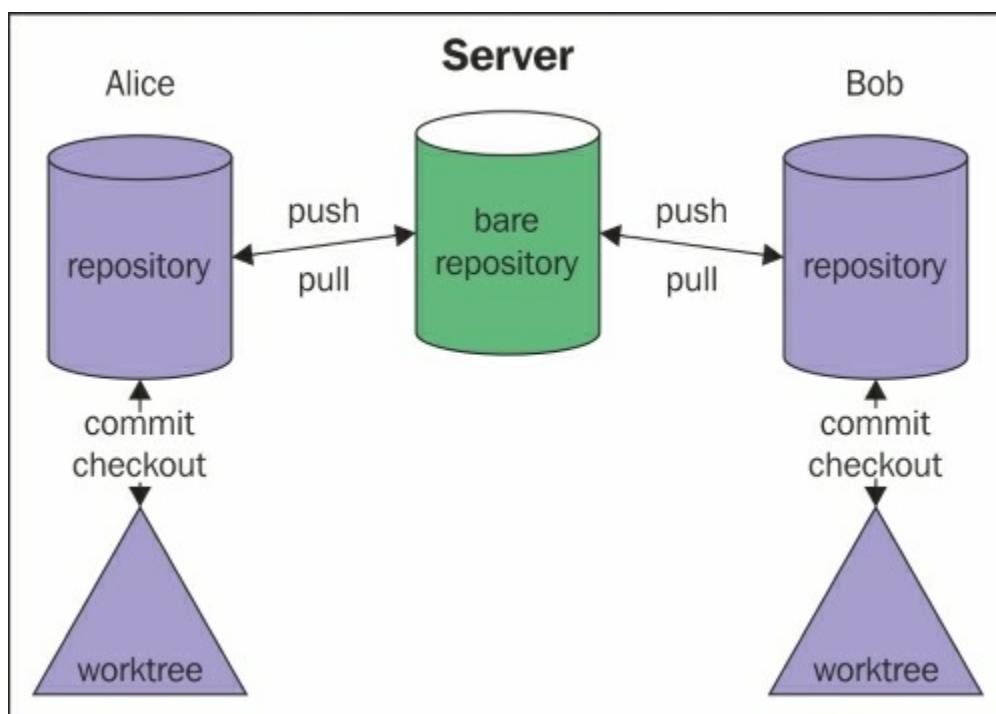
Git by example

Let's follow step by step a simple example of two developers using Git to work together on a simple project. You can download the example code files from <http://www.packtpub.com>. You can find there all three repositories (for two developers, and the bare server repository) with the example code files for this chapter, where you can examine code, history, and reflog..

Repository setup

A company has begun work on a new product. This product calculates a random number—an integer value of specified range.

The company has assigned two developers to work on this new project, Alice and Bob. Both developers are telecommuting to the company's corporate headquarters. After a bit of discussion, they have decided to implement their product as a command-line app in C, and to use Git 2.5.0 (<http://git-scm.com/>) for version control. This project and the code are intended for demonstration purposes, and will be much simplified. The details of code are not important here—what's important is how the code changes:



With a small team, they have decided on the setup shown in the preceding diagram.

Note

This is one possible setup, with the central *canonical* repository, and without a dedicated maintainer responsible for this repository (all developers are equal in this setup). It is not the only possibility; other ways of configuring repositories will be shown in [Chapter 5, Collaborative Development with Git](#).

Creating a Git repository

Alice gets the project started by asking Carol, an administrator, to create a new repository specifically for collaborating on a project, to share work with all the team:

Note

Command line examples follow the Unix convention of having `user@host` and directory in the command prompt, to know from the first glance who performs a command, on what computer, and in which directory. This is the usual setup on Unix (for example, on Linux).

You can configure your command prompt to show Git-specific information like the name of the repository name, the subdirectory within the repository, the current branch, and even the status of the working area, see [Chapter 10, Customizing and Extending Git](#).

```
carol@server ~$ mkdir -p /srv/git
carol@server ~$ cd /srv/git
carol@server /srv/git$ git init --bare random.git
```

Note

I consider the details of server configuration to be too much for this chapter. Just imagine that it happened, and nothing went wrong. Or take a look at [Chapter 11, Git Administration](#).

You can also use a tool to manage Git repositories (for example Gitolite); creating a public repository on a server would then, of course, look different. Often though it involves creating a Git repository with `git init` (without `--bare`) and then pushing it with an explicit URI to the server, which then automatically creates the public repository.

Or perhaps the repository was created through the web interface of tools, such as GitHub, Bitbucket, or GitLab (either hosted or on-premise).

Cloning the repository and creating the first commit

Bob gets the information that the project repository is ready, and he can start coding.

Since this is Bob's first time using Git, he first sets up his `~/.gitconfig` file with information that will be used to identify his commits in the log:

```
[user]
  name = Bob Hacker
  email = bob@company.com
```

Now he needs to get his own repository instance:

```
bob@hostB ~$ git clone https://git.company.com/random
Cloning into random...
Warning: You appear to have cloned an empty repository.
done.
bob@hostB ~$ cd random
bob@hostB random$
```

Note

All examples in this chapter use the command-line interface. Those commands might be given using a Git GUI or IDE integration. The *Git: Version Control for Everyone* book, published by Packt Publishing, shows GUI equivalents for the command-line.

Bob notices that Git said that it is an empty repository, with no source

code yet, and starts coding. He opens his text editor and creates the starting point for their product:

```
#include <stdio.h>
#include <stdlib.h>

int random_int(int max)
{
    return rand() % max;
}

int main(int argc, char *argv[])
{
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <number>\n", argv[0]);
        return EXIT_FAILURE;
    }

    int max = atoi(argv[1]);

    int result = random_int(max);
    printf("%d\n", result);

    return EXIT_SUCCESS;
}
```

Typically, like for most initial implementations, this version is missing a lot of features. But it's a good place to begin. Before committing his code, Bob wants to make sure that it compiles and runs:

```
bob@hostB random$ gcc -std=c99 random.c
bob@hostB random$ ls -l
total 43
-rwxr-xr-x 1 bob    staff  86139 May 29 17:36 a.out
-rw-r--r-- 1 bob    staff     331 May 19 17:11 random.c
bob@hostB random$ ./a.out
Usage: ./a.out <number>
bob@hostB random$ ./a.out 10
1
```

Alright! It's time to add this file to the repository:

```
bob@hostB random$ git add random.c
```

Bob uses the status operation to make sure that the pending changeset (the future commit) looks proper:

Note

We use here a short form of the `git status` to reduce the amount of space taken by examples; you can find an example of full status output further in the chapter.

```
bob@hostB random$ git status -s
A  random.c
?? a.out
```

Git is complaining because it does not know what to do about the `a.out` file: it is neither tracked nor ignored. That's a compiled executable, which as a generated file should not be stored in a version control repository. Bob can just ignore that issue for the time being.

Now it's time to `commit` the file:

```
bob@hostB random$ git commit -a -m "Initial implementation"
[master (root-commit) 2b953b4] Initial implementation
 1 file changed, 22 insertions(+)
 Create mode 100644 random.c
```

Note

Normally, you would create a `commit` message not by using the `-m <message>` command-line option, but by letting Git open an editor. We use this form here to make examples more compact.

The `-a` / `--all` option means to take all changes to the tracked files; you can separate manipulating the staging area from creating a commit —this is however a separate issue, left for [Chapter 4, Managing Your Worktree](#).

Publishing changes

After finishing working on the initial version of the project, Bob decides that it is ready to be published (to be made available for other

developers). He pushes the changes:

```
bob@hostB random$ git push
warning: push.default is unset; its implicit value has changed
in
Git 2.0 from 'matching' to 'simple'. To squelch this message
[...]
To https://git.company.com/random
 * [new branch]      master -> master
bob@hostB random$ git config --global push.default simple
```

Note

Note that, depending on the speed of network, Git could show progress information during remote operations such as `clone`, `push`, and `fetch`. Such information is omitted from examples in this book, except where that information is actually discussed while examining history and viewing changes.

Examining history and viewing changes

Since it is Alice's first time using Git on her desktop machine, she first tells Git how her commits should be identified:

```
alice@hostA ~$ git config --global user.name "Alice Developer"
alice@hostA ~$ git config --global user.email alice@company.com
```

Now Alice needs to set up her own repository instance:

```
alice@hostA ~$ git clone https://git.company.com/random
Cloning into random...
done.
```

Alice examines the working directory:

```
alice@hostA ~$ cd random
alice@hostA random$ ls -al
total 1
drwxr-xr-x    1 alice staff          0 May 30 16:44 .
drwxr-xr-x    4 alice staff          0 May 30 16:39 ..
drwxr-xr-x    1 alice staff          0 May 30 16:39 .git
-rw-r--r--    1 alice staff        353 May 30 16:39 random.c
```

Note

The `.git` directory contains Alice's whole copy (clone) of the repository in Git internal format, and some repository-specific administrative information. See `gitrepository-layout(5)` manpage for details of the file layout, which can be done for example with `git help repository-layout` command.

She wants to check the `log` to see the details (to examine the project history):

```
alice@hostA random$ git log
commit 2b953b4e80abfb77bdcd94e74dedeeebf6aba870
Author: Bob Hacker <bob@company.com>
Date:   Thu May 29 19:53:54 2015 +0200

    Initial implementation
```

Note

Naming revisions:

At the lowest level, a Git version identifier is a SHA-1 hash, for example `2b953b4e80`. Git supports various forms of referring to revisions, including the unambiguously shortened SHA-1 (with a minimum of four characters)—see [Chapter 2, Exploring Project History](#), for more ways.

When Alice decides to take a look at the code, she immediately finds something horrifying. The random number generator is never initialized! A quick test shows that the program always generates the same number. Fortunately, it is only necessary to add one line to `main()`, and the appropriate `#include`:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int random_int(int max)
{
    return rand() % max;
}
```

```

int main(int argc, char *argv[])
{
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <number>\n", argv[0]);
        return EXIT_FAILURE;
    }

    int max = atoi(argv[1]);

    srand(time(NULL));
    int result = random_int(max);
    printf("%d\n", result);

    return EXIT_SUCCESS;
}

```

She compiles the code, and runs it a few times to check that it really generates random numbers. Everything looks alright, so she uses the status operation to see the pending changes:

```
alice@hostA random$ git status -s
M random.c
```

No surprise here. Git knows that `random.c` has been modified. She wants to double-check by reviewing the actual changes with the `diff` command:

Note

From here on, we will not show untracked files, unless they are relevant to the topic being discussed; let's assume that Alice set up an ignore file, as described in [Chapter 4, Managing Your Worktree](#).

```
alice@hostA random$ git diff
diff --git a/random.c b/random.c
index cc09a47..5e095ce 100644
--- a/random.c
+++ b/random.c
@@ -1,5 +1,6 @@
 #include <stdio.h>
 #include <stdlib.h>
+#include <time.h>
```

```

int random_int(int max)
{
@@ -15,6 +16,7 @@ int main(int argc, char *argv[])
    int max = atoi(argv[1]);
+    srand(time(NULL));
    int result = random_int(max);
    printf("%d\n", result);

```

Now it's time to commit the changes and push them to the public repository:

```

alice@hostA random$ git commit -a -m "Initialize random number generator"
[master db23d0e] Initialize random number generator
 1 file changed, 2 insertions(+)
alice@hostA random$ git push
To https://git.company.com/random
 3b16f17..db23d0e master -> masterRenaming and moving files

```

Renaming and moving files

Bob moves on to his next task, which is to restructure the tree a bit. He doesn't want the top level of the repository to get too cluttered so he decides to move all their source code files into a `src/` subdirectory:

```

bob@hostA random$ mkdir src
bob@hostA random$ git mv random.c src/
bob@hostA random$ git status -s
R random.c -> src/random.c
bob@hostA random$ git commit -a -m "Directory structure"
[master 69e0d3d] Directory structure
 1 file changed, 0 insertions(+), 0 deletions(-)
 rename random.c => src/random.c (100%)

```

While at it, to minimize the impact of reorganization on the `diff` output, he configures Git to always use rename and copy detection:

```
bob@hostB random$ git config --global diff.renames copies
```

Bob then decides the time has come to create a proper `Makefile`, and to add a `README` for a project:

```
bob@hostA random$ git add README Makefile
bob@hostA random$ git status -s
A  Makefile
A  README
bob@hostA random$ git commit -a -m "Added Makefile and README"
[master abfeeaa] Added Makefile and README
 2 files changed, 15 insertions(+)
 create mode 100644 Makefile
 create mode 100644 README
```

Bob decides to rename `random.c` to `rand.c`:

```
bob@hostA random$ git mv src/random.c src/rand.c
```

This of course also requires changes to the Makefile:

```
bob@hostA random$ git status -s
M Makefile
R  src/random.c -> src/rand.c
```

He then commits those changes.

Updating your repository (with merge)

Reorganization done, now Bob tries to publish those changes:

```
bob@hostA random$ git push
$ git push
To https://git.company.com/random
 ! [rejected]      master -> master (fetch first)
error: failed to push some refs to
'https://git.company.com/random'
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.
```

But Alice was working at the same time and she had her change ready to commit and push first. Git is not allowing Bob to publish his changes

because Alice has already pushed something to the master branch, and Git is preserving her changes.

Note

Hints and advices in Git command output will be skipped from here on for the sake of brevity.

Bob uses `pull` to bring in changes (as described in `hint` in the command output):

```
bob@hostB random $ git pull
From https://git.company.com/random
 + 3b16f17...db23d0e master      -> origin/master
Auto-merging src/rand.c
Merge made by the 'recursive' strategy.
 src/rand.c | 2 ++
 1 file changed, 2 insertions(+)
```

Git `pull` fetched the changes, automatically merged them with Bob's changes, and committed the merge.

Everything now seems to be good:

```
bob@hostB random$ git show
commit ba5807e44d75285244e1d2eachb1c10cbc5cf3935
Merge: 3b16f17 db23d0e
Author: Bob Hacker <bob@company.com>
Date:   Sat May 31 20:43:42 2015 +0200

Merge branch 'master' of https://git.company.com/random
```

The merge commit is done. Apparently, Git was able to merge Alice's changes directly into Bob's moved and renamed copy of a file without any problems. Marvelous!

Bob checks that it compiles (because *automatically merged* does not necessarily mean that the merge output is correct), and is ready to push the merge:

```
bob@hostB random$ git push
To https://git.company.com/random
```

```
db23d0e..ba5807e master -> master
```

Creating a tag

Alice and Bob decide that the project is ready for wider distribution. Bob creates a tag so they can more easily access/refer to the released version. He uses an **annotated tag** for this; an often used alternative is to use **signed tag**, where the annotation contains a PGP signature (which can later be verified):

```
bob@hostB random$ git tag -a -m "random v0.1" v0.1
bob@hostB random$ git tag --list
v0.1
bob@hostB random$ git log -1 --decorate --abbrev-commit
commit ba5807e (HEAD -> master, tag: v0.1, origin/master)
Merge: 3b16f17 db23d0e
Author: Bob Hacker <bob@company.com>
Date:   Sat May 31 20:43:42 2015 +0200

Merge branch 'master' of https://git.company.com/random
```

Of course, the v0.1 tag wouldn't help if it was only in Bob's local repository. He therefore pushes the just created tag:

```
bob@hostB random$ git push origin tag v0.1
Counting objects: 1, done.
Writing objects: 100% (1/1), 162 bytes, done.
Total 1 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (1/1), done.
To https://git.company.com/random
 * [new tag]           v0.1 -> v0.1
```

Alice updates her repository to get the v0.1 tag, and to start with up-to-date work:

```
alice@hostA random$ git pull
From https://git.company.com/random
 f4d9753..be08dee master      -> origin/master
 * [new tag]           v0.1      -> v0.1
Updating f4d9753..be08dee
Fast-forward
 Makefile               | 11 ++++++
 README                |  4 +++

```

```
random.c => src/rand.c | 0
3 files changed, 15 insertions(+)
create mode 100644 Makefile
create mode 100644 README
rename random.c => src/rand.c (100%)
```

Resolving a merge conflict

Alice decides that it would be a good idea to extract initialization of a pseudo-random numbers generator into a separate subroutine. This way, both initialization and generating random numbers are encapsulated, making future changes easier. She codes and adds `init_rand()`:

```
void init_rand(void)
{
    srand(time(NULL));
}
```

Grand! Let's see that it compiles.

```
alice@hostA random$ make
gcc -std=c99 -Wall -Wextra -o rand src/rand.c
alice@hostA random$ ls -F
Makefile  rand*  README  src/
```

Good. Time to commit the change:

```
alice@hostA random$ git status -s
M src/rand.c
alice@hostA random$ git commit -a -m "Abstract RNG
initialization"
[master 26f8e35] Abstract RNG initialization
1 files changed, 6 insertions(+), 1 deletion(-)
```

No problems here.

Meanwhile, Bob notices that the documentation for the `rand()` function used says that it is a weak pseudo-random generator. On the other hand, it is a standard function, and it might be enough for the planned use:

```
bob@hostB random$ git pull
Already up-to-date.
```

He decides to add a note about this issue in a comment:

```
bob@hostB random$ git status -s
 M src/rand.c
bob@hostB random$ git diff
diff --git a/src/rand.c b/src/rand.c
index 5e095ce..8fddf5d 100644
--- a/src/rand.c
+++ b/src/rand.c
@@ -2,6 +2,7 @@
 #include <stdlib.h>
 #include <time.h>

+// TODO: use a better random generator
 int random_int(int max)
 {
     return rand() % max;
```

He has his change ready to commit and push first:

```
bob@hostB random$ git commit -m 'Add TODO comment for
random_int()'
[master 8c4ceca] Use Add TODO comment for random_int()
 1 files changed, 1 insertion(+)
bob@hostB random$ git push
To https://git.company.com/random
 ba5807e..8c4ceca master -> master
```

So when Alice is ready to push her changes, Git rejects it:

```
alice@hostA random$ git push
To https://git.company.com/random
 ! [rejected]          master -> master (non-fast-forward)
error: failed to push some refs to
'https://git.company.com/random'
[...]
```

Ah. Bob must have pushed a new changeset already. Alice once again needs to pull and merge to combine Bob's changes with her own:

```
alice@hostA random$ git pull
From https://git.company.com/random
 ba5807e..8c4ceca master      -> origin/master
Auto-merging src/rand.c
CONFLICT (content): Merge conflict in src/rand.c
```

Automatic merge failed; fix conflicts and then commit the result.

The merge didn't go quite as smoothly this time. Git wasn't able to automatically merge Alice's and Bob's changes. Apparently, there was a conflict. Alice decides to open the `src/rand.c` file in her editor to examine the situation (she could have used a graphical merge tool via `git mergetool` instead):

```
<<<<<< HEAD
void init_rand(void)
{
    srand(time(NULL)) ;
}

=====
// TODO: use a better random generator
>>>>> 8c4ceca59d7402fb24a672c624b7ad816cf04e08
int random_int(int max)
```

Git has included both Alice's code (between `<<<<<< HEAD` and `=====` conflict markers) and Bob's code (between `=====` and `>>>>>>`). What we want as a final result is to include both blocks of code. Git couldn't merge it automatically because those blocks were not separated. Alice's `init_rand()` function can be simply included right before Bob's added comment. After resolution, the changes look like this:

```
alice@hostA random$ git diff
diff --cc src/rand.c
index 17ad8ea,8fddf5d..0000000
--- a/src/rand.c
+++ b/src/rand.c
@@@ -2,11 -2,7 +2,12 @@@
 #include <stdlib.h>
 #include <time.h>

+void init_rand(void)
+{
+    srand(time(NULL)) ;
+
+
+ // TODO: use a better random generator
```

```
int random_int(int max)
{
    return rand() % max;
```

That should take care of the problem. Alice compiles the code to make sure and then commits the merge:

```
alice@hostA random$ git status -s
UU src/rand.c
alice@hostA random$ git commit -a -m 'Merge: init_rand() +
TODO'
[master 493e222] Merge: init_rand() + TODO
```

And then she retries the push:

```
alice@hostA random$ git push
To https://git.company.com/random
  8c4ceca..493e222  master -> master
```

And... done.

Adding files in bulk and removing files

Bob decides to add a `COPYRIGHT` file with a copyright notice for the project. There was also a `NEWS` file planned (but not created), so he uses a bulk `add` to add all the files:

```
bob@hostB random$ git add -v
add 'COPYRIGHT'
add 'COPYRIGHT~'
```

Oops. Because Bob didn't configure his **ignore patterns**, the backup file `COPYRIGHT~` was caught too. Let's remove this file:

```
bob@hostB random$ git status -s
A  COPYRIGHT
A  COPYRIGHT~
bob@hostB random$ git rm COPYRIGHT~
error: 'COPYRIGHT~' has changes staged in the index
(use --cached to keep the file, or -f to force removal)
bob@hostB random$ git rm -f COPYRIGHT~
rm 'COPYRIGHT~'
```

Let's check the status and commit the changes:

```
bob@hostB random$ git status -s
A  COPYRIGHT
bob@hostB random$ git commit -a -m 'Added COPYRIGHT'
[master ca3cdd6] Added COPYRIGHT
 1 files changed, 2 insertions(+), 0 deletions(-)
 create mode 100644 COPYRIGHT
```

Undoing changes to a file

A bit bored, Bob decides to indent `rand.c` to make it follow a consistent coding style convention:

```
bob@hostB random$ indent src/rand.c
```

He checks how much source code it changed:

```
bob@hostB random$ git diff --stat
src/rand.c |    40 ++++++-----+
1 files changed, 22 insertions(+), 18 deletions(-)
```

That's too much (for such a short file). It could cause problems with merging. Bob calms down and undoes the changes to `rand.c`:

```
bob@hostB random$ git status -s
 M src/rand.c
bob@hostB random$ git checkout -- src/rand.c
bob@hostB random$ git status -s
```

Note

If you don't remember how to revert a particular type of change, or to update what is to be committed (using `git commit` without `-a`), the output of `git status` (without `-s`) contains information about what commands to use. This is shown as follows:

```
bob@hostB random$ git status
# On branch master
# Your branch is ahead of 'origin/master' by 1 commit.
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
```

```
#   (use "git checkout -- <file>..." to discard changes in  
working directory)  
#  
# modified:    src/rand.c
```

Creating a new branch

Alice notices that using a modulo operation to return random numbers within a given span does not generate uniformly distributed random numbers, since in most cases it makes lower numbers slightly more likely. She decides to try to fix this issue. To isolate this line of development from other changes, she decides to create her own named branch (see also [Chapter 6, Advanced Branching Techniques](#)), and switch to it:

```
alice@hostA random$ git checkout -b better-random  
Switched to a new branch 'better-random'  
alice@hostA random$ git branch  
* better-random  
  master
```

Note

Instead of using the `git checkout -b better-random` shortcut to create a new branch and switch to it in one command invocation, she could have first created a branch with `git branch better-random`, then switched to it with `git checkout better-random`.

She decides to shrink the range from `RAND_MAX` to the requested number by rescaling the output of `rand()`. The changes look like this:

```
alice@hostA random$ git diff  
diff --git a/src/rand.c b/src/rand.c  
index 2125b0d..5ded9bb 100644  
--- a/src/rand.c  
+++ b/src/rand.c  
@@ -10,7 +10,7 @@ void init_rand(void)  
 // TODO: use a better random generator  
 int random_int(int max)  
{  
-     return rand() % max;  
+     return rand()*max / RAND_MAX;
```

```
}
```

```
int main(int argc, char *argv[])
```

She commits her changes, and pushes them, knowing that the push will succeed because she is working on her private branch:

```
alice@hostA random$ git commit -a -m 'random_int: use  
rescaling'  
[better-random bb71a80] random_int: use rescaling  
 1 files changed, 1 insertion(+), 1 deletion(-)  
alice@hostA random$ git push  
fatal: The current branch better-random has no upstream branch.  
To push the current branch and set the remote as upstream, use  
  
    git push --set-upstream origin better-random
```

Alright! Git just wants Alice to set up a remote origin as the upstream for the newly created branch (it is using a simple push strategy); this will also push this branch explicitly.

```
alice@hostA random$ git push --set-upstream origin better-  
random  
To https://git.company.com/random  
 * [new branch]      better-random -> better-random
```

Note

If she wants to make her branch visible, but private (so nobody but her can push to it), she needs to configure the server with hooks, or use Git repository management software such as `Gitolite` to manage it for her.

Merging a branch (no conflicts)

Meanwhile, over in the default branch, Bob decides to push his changes by adding the `COPYRIGHT` file:

```
bob@hostB random$ git push  
To https://git.company.com/random  
 ! [rejected]      master -> master (non-fast-forward)  
[...]
```

OK. Alice was busy working at extracting random number generator

initialization into a subroutine (and resolving a merge conflict), and she pushed the changes first:

```
bob@hostB random$ git pull
From https://git.company.com/random
  8c4ceca..493e222  master      -> origin/master
 * [new branch]      better-random -> origin/better-random
Merge made by 'recursive' strategy.
 src/rand.c | 7 ++++++-
 1 file changed, 6 insertions(+), 1 deletion(-)
```

Well, Git has merged Alice's changes cleanly, but there is a new branch present. Let's take a look at what is in it, showing only those changes exclusive to the `better-random` branch (the double dot syntax is described in [Chapter 2, Exploring Project History](#)):

```
bob@hostB random$ git log HEAD..origin/better-random
commit bb71a804f9686c4bada861b3fc3cfb5600d2a47
Author: Alice Developer <alice@company.com>
Date:   Sun Jun 1 03:02:09 2015 +0200
```

```
random_int: use rescaling
```

Interesting. Bob decides he wants that. So he asks Git to merge stuff from Alice's branch (which is available in the respective remote-tracking branch) into the default branch:

```
bob@hostB random$ git merge origin/better-random
Merge made by the 'recursive' strategy.
 src/rand.c | 2 ++
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Undoing an unpublished merge

Bob realizes that it should be up to Alice to decide when the feature is ready for inclusion. He decides to undo a merge. Because it is not published, it is as simple as rewinding to the previous state of the current branch:

```
bob@hostB random$ $ git reset --hard @{1}
HEAD is now at 3915cef Merge branch 'master' of
https://git.company.com/random
```

Note

This example demonstrates the use of reflog for undoing operations; another solution would be to go to a previous (pre-merge) commit following the first parent, with `HEAD^` instead of `@{1}`.

Summary

This chapter walked us through the process of working on a simple example project by a small development team.

We have recalled how to start working with Git, either by creating a new repository or by cloning an existing one. We have seen how to prepare a commit by adding, editing, moving, and renaming files, how to revert changes to file, how to examine the current status and view changes to be committed, and how to tag a new release.

We have recalled how to use Git to work at the same time on the same project, how to make our work public, and how to get changes from other developers. Though using a version control system helps with simultaneous work, sometimes Git needs user input to resolve conflicts in work done by different developers. We have seen how to resolve a merge conflict.

We have recalled how to create a tag marking a release, and how to create a branch starting an independent line of development. Git requires tags and new branches to be pushed explicitly, but it fetches them automatically. We have seen how to merge a branch.

Chapter 2. Exploring Project History

One of the most important parts of mastering a version control system is exploring project history, making use of the fact that with version control systems we have an archive of every version that has ever existed. Here, the reader will learn how to select, filter, and view the range of revisions; how to refer to the revisions (revision selection); and how to find revisions using different criteria.

This chapter will introduce the concept of **Directed Acyclic Graph (DAG)** of revisions and explain how this concept relates to the ideas of branches, tags, and of the current branch in Git.

Here is the list of topics we will cover in this chapter:

- Revision selection
- Revision range selection, limiting history, history simplification
- Searching history with "pickaxe" tool and diff search
- Finding bugs with `git bisect`
- Line-wise history of file contents with `git blame`, and rename detection
- Selecting and formatting output (the `pretty` formats)
- Summarizing contribution with `shortlog`
- Specifying canonical author name and e-mail with `.mailmap`
- Viewing specific revision, diff output options, and viewing file at revision

Directed Acyclic Graphs

What makes version control systems different from backup applications is, among others, the ability to represent more than linear history. This is necessary, both to support the simultaneous parallel development by different developers (each developer in his or her own clone of repository), and to allow independent parallel lines of development—

branches. For example, one might want to keep the ongoing development and work on bug fixes for the stable version isolated; this is possible by using individual branches for the separate lines of development. **Version control system (VCS)** thus needs to be able to model such a (non-linear) way of development and to have some structure to represent multiple revisions.

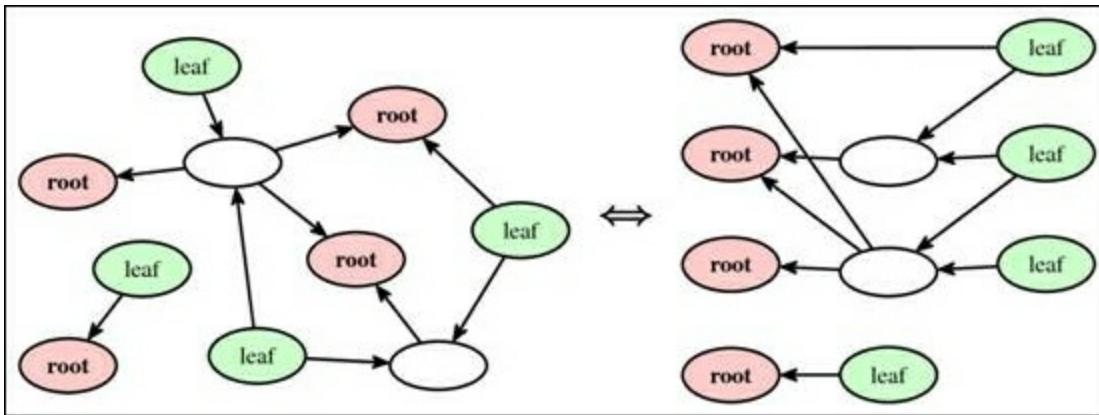


Fig 1. A generic example of the Directed Acyclic Graph (DAG). The same graph is represented on both sides: in free-form on the left, left-to-right order on the right.

The structure that Git uses (on the abstract level) to represent the possible non-linear history of a project is called a Directed Acyclic Graph (DAG).

A *directed graph* is a data structure from computer science (and mathematics) composed of nodes (vertices) that are connected with directed edges (arrows). A directed graph is *acyclic* if it doesn't contain any cycles, which means that there is no way to start at some node and follow a sequence of the directed edges to end up back at the starting node.

In concrete examples of graphs, each node represents some object or a piece of data, and each edge from one node to another represents some kind of relationship between objects or data, represented by the nodes this edge connects.

The DAG of revisions in **distributed version control systems (DVCS)** uses the following representation:

- **Nodes:** In DVCS, each node represents one revision (one version) of a project (of the entire tree). These objects are called **commits**.
- **Directed edges:** In DVCS, each edge is based on the relationship between two revisions. The arrow goes from a later **child** revision to an earlier **parent** revision it was based on or created from.

As directed edges' representation is based on a causal relationship between revisions, the arrows in the DAG of revisions may not form a cycle. Usually, the DAG of revisions is laid out left-to-right (root nodes on the left, leaves on the right) or bottom-to-top (the most recent revisions on top). Figures in this book and ASCII-art examples in Git documentation use the left-to-right convention, while the Git command line use bottom-to-top, that is, the most recent first convention.

There are two special type of nodes in any DAG (see *Fig 1*):

- **Root nodes:** These are the nodes (revisions) that have no parents (no outgoing edges). There is at least one root node in the DAG of revisions, which represents the initial (starting) version of a project.

Note

There can be more than one root node in Git's DAG of revisions. Additional root nodes can be created when joining two formerly originally independent projects together; each joined project brings its own root node.

Another source of root nodes are **orphan branches**, that is, disconnected branches having no history in common. They are, for example, used by GitHub to manage a project's web pages together in one repository with code, and by Git project to store the pregenerated documentation (the `man` and `html` branches) or related projects (`todo`).

- **Leaf nodes (or leaves):** These are the nodes that have no children (no incoming edges); there is at least one such node. They represent

the most recent versions of the project, not having any work based on them. Usually, each leaf in the DAG of revisions has a branch head pointing to it.

The fact that the DAG can have more than one leaf node means that there is no inherent notion of the latest version, as it was in the linear history paradigm.

Whole-tree commits

In DVCS, each node of the DAG of revisions (a model of history) represents a version of the project as a whole single entity: of all the files and all the directories, and of the whole directory tree of a project.

This means that each developer will always get the history of all the files in his or her clone of the repository. He or she can choose to get only a part of the history (shallow clone and/or cloning only selected branches) and checkout only the selected files (sparse checkout), but to date, there is no way to get only the history of the selected files in the clone of the repository. [Chapter 9, Managing Subprojects - Building a Living Framework](#) will show some workarounds for when you want to have the equivalent of the partial clone, for example, when working with large media files that are needed only for a selected subset of your developers.

Branches and tags

A **branch operation** is what you use when you want your development process to fork into two different directions to create another line of development. For example, you might want to create a separate branch to keep managing bug fixes to the released stable version, isolating this activity from the rest of the development.

A **tag operation** is a way to associate a meaningful symbolic name with the specific revision in the repository. For example, you might want to create v1.3-rc3 with the third release candidate before releasing version 1.3 of your project . This makes it possible to go back to this specific

version, for example, to check the validity of the bug report.

Both branches and tags, sometimes called **references (refs)** together, have the same meaning (the same representation) within the DAG of revisions. They are the external references (pointers) to the graph of revisions, as shown in *Fig 2*.

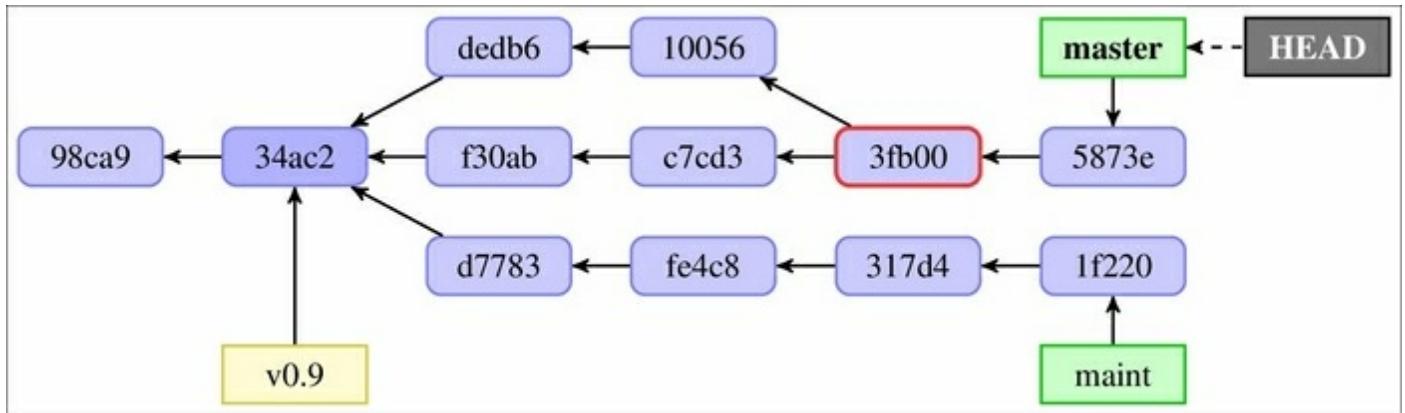


Fig 2. Example graph of revisions in a version control system, with two branches "master" (current branch) and "maint", single tag "v0.9", one branching point with shortened identifier 34ac2, and one merge commit: 3fb00.

A **tag** is a symbolic name (for example, `v1.3-rc3`) for a given revision. It always points to the same object; it does not change. The idea behind having tags is, for every project's developer, to be able to refer to the given revision with a symbolic name, and to have this symbolic name mean the same for each and every developer. Checking out or viewing the given tag should have the same results for everyone.

A **branch** is a symbolic name for the line of development. The most recent commit (leaf revision) on such a line of development is referred to as the top or tip of the branch, or branch head, or just a branch. Creating a new commit will generate a new node in the DAG, and advance the appropriate branch ref.

The branch in the DAG is, as a line of development, the subgraph of the revisions composed of those revisions that are reachable from the tip of

the branch (the branch head); in other words, revisions that you can walk to by following the *parent* edges starting from the branch head.

Git, of course, needs to know which branch tip to advance when creating a new commit. It needs to know which branch is the current one and is checked out into the working directory. Git uses the **HEAD** pointer for this, as shown in *Fig 2* of this chapter. Usually, this points to one of branch tips, which, in turn, points to some node in the DAG of revisions, but not always—see [Chapter 3, Developing with Git](#), for an explanation of the **detached HEAD** situation; that is, when HEAD points directly to a node in the DAG.

Note

Full names of references (branches and tags)

Originally, Git stored branches and tags in files inside `.git` administrative area, in the `.git/refs/heads/` and `.git/refs/tags/` directories, respectively. Modern Git can store information about tags and branches inside the `.git/packed-refs` file to avoid handling a very large number of small files. Nevertheless, active references use original *loose* format—one file per reference.

The `HEAD` pointer (usually a symbolic reference, for example `ref: refs/heads/master`) is stored in `.git/HEAD`.

The `master` branch is stored in `.git/refs/heads/master`, and has `refs/heads/master` as full name (in other words, branches reside in the `refs/heads/` namespace). The tip of the branch is referred to as *head* of a branch, hence the name of a namespace. In *loose* format, the file content is an SHA-1 identifier of the most current revision on the branch (the *branch tip*), in plain text as hexadecimal digit. It is sometimes required to use the full name if there is ambiguity among refs.

The remote-tracking branch, `origin/master`, which remembers the last seen position of the `master` branch in the remote repository, `origin`, is stored in `.git/refs/remotes/origin/master`, and has

`refs/remotes/origin/master` as its full name. The concept of **remotes** will be explained in [Chapter 5, Collaborative Development with Git](#), and that of **remote-tracking branches** in [Chapter 6, Advanced Branching Techniques](#).

The `v1.3-rc3` tag has `refs/tags/v1.3-rc3` as the full name (tags reside in the `refs/tags/` namespace). To be more precise, in the case of **annotated** and **signed tags**, this file stores references to the **tag object**, which, in turn, points to the node in the DAG, and not directly to a commit. This is the only type of ref that can point to any type of object.

These full names (fully qualified names) can be seen when using commands intended for scripts, for example, `git show-ref`.

Branch points

When you create a new branch starting at a given version, the lines of development usually diverge. The act of creating a divergent branch is denoted in the DAG by a commit, which has more than one child, that is a node pointed to by more than one arrow.

Note

Git does not track information about creating (forking) a branch, and does not mark branch points in any way that is preserved across clones and pushes. There is information about this event in the reflog (*branch created from HEAD*), but this is local to the repository where branching occurred, and is temporary. However, if you know that the `B` branch started from the `A` branch, you can find a branching point with `git merge-base A B`; in modern Git you can use `--fork-point` option to make it also use the reflog.

In *Fig 2*, the commit `34ac2` is a branching point for the **master** and **maint** branches.

Merge commits

Typically, when you have used branches to enable independent parallel

development, you will later want to join them. For example, you would want bug fixes applied to the stable (maintenance) branch to be included in the main line of development as well (if they are applicable and were not fixed accidentally during the main-line development).

You would also want to merge changes created in parallel by different developers working simultaneously on the same project, each using their own clone of repository and creating their own lines of commits.

Such a merge operation will create a new revision, joining two lines of development. The result of this operation will be based on more than one commit. A node in the DAG representing the said revision will have more than one parent. Such an object is called a merge commit.

You can see a merge commit, **3fb00**, in *Fig 2*.

Single revision selection

During development, many times you want to select a single revision in the history of a project, to examine it, or to compare with the current version. The ability to select revision is also the basis for selecting a revision range, for example a subsection of history to examine.

Many Git commands take revision parameters as arguments, which is typically denoted by `<rev>` in Git reference documentation. Git allows you to specify specific commits or a range of commits in several ways.

HEAD – the implicit revision

Most, but not all, Git commands that require the revision parameter, default to using `HEAD`. For example, `git log` and `git log HEAD` will show the same information.

The `HEAD` denotes the *current branch*, or in other words the commit that was checked out into the working directory, and forms a base of a current work.

There are a few other references which are similar to `HEAD`:

- `FETCH_HEAD`: This records the information about the remote branches that were fetched from a remote repository with your last `git fetch` or `git pull` invocation. It is very useful for one-off fetch, with the repository to fetch from given by a URL, unlike when fetching from a named repository such as `origin`, where we can use the remote-tracking branch instead, for example, `origin/master`. Moreover, with named repositories, we can use the reflog for remote-tracking branch, for example, `origin/master@{1}`, to get the position before the fetch. Note that `FETCH_HEAD` is overwritten by each fetch from any repository.
- `ORIG_HEAD`: This records the previous position of the current branch; this reference is created by commands that move the current branch in a drastic way (creating a new commit doesn't set `ORIG_HEAD`) to record the position of the `HEAD` command before the operation. It is

very useful if you want to undo or abort such operations; though nowadays the same can be done using reflogs, which store additional information that can be examined in their use.

You can also stumble upon the short-lived temporary references used during specific operations:

- During a merge, before creating a merge commit, the `MERGE_HEAD` records the commit(s) that you are merging into your branch. It vanishes after creating a merge commit.
- During a cherry-pick, before creating a commit that copies picked changes into another branch, the `CHERRY_PICK_HEAD` records the commit that you have selected for cherry-picking.

Branch and tag references

The most straightforward and commonly used way to specify a revision is to use symbolic names: branches, naming the line of development, pointing to the tip of said line, and tags, naming specific revision. This way of specifying revisions can be used to view the history of a line of development, examine the most current revision (current work) on a given branch, or compare the branch or tag with the current work.

You can use any refs (external references to the DAG of revisions) to select a commit. You can use a branch name, tag name, and remote-tracking branch in any Git command that requires revision as a parameter.

Usually, it is enough to give a *short* name to a branch or tag, for example, `git log master`, to view the history of a `master` branch, or `git log v1.3-rc3` to see how version `v1.3-rc1` came about. It can, however, happen that there are different types of refs with the same name, for example, both branch and tag named `dev` (though it is recommended to avoid such situations). Or, you could have created (usually by accident) the local branch, `origin/master`, when there was a remote-tracking branch with the short name, `origin/master`, tracking where the `master` branch was in the remote repository, `origin`.

In such a situation, when ref name is ambiguous, it is disambiguated by taking the first match in the following rules (this is a shortened and simplified version; for the full list, see the *gitrevisions(7)* manpage):

1. The top level symbolic name, for example, HEAD.
2. Otherwise, the name of the tag (refs/tags/ namespace).
3. Otherwise, the name of the local branch (refs/heads/ namespace).
4. Otherwise, the name of the remote-tracking branch (refs/remotes/ namespace).
5. Otherwise, the name of the remote if there exists a default branch for it; the revision is said default branch (example refs/remotes/origin/HEAD for origin as a parameter).

SHA-1 and the shortened SHA-1 identifier

In Git, each revision is given a unique identifier (object name), which is a SHA-1 hash function, based on the contents of the revision. You can select a commit by using its SHA-1 identifier as a 40-character long hexadecimal number (160 bits). Git shows full SHA-1 identifiers in many places, for example, you can find them in the full `git log` output:

```
$ git log
commit 50f84e34a1b0bb893327043cb0c491e02ced9ff5
Author: Junio C Hamano <gitster@pobox.com>
Date:   Mon Jun 9 11:39:43 2014 -0700

    Update draft release notes to 2.1

    Signed-off-by: Junio C Hamano <gitster@pobox.com>

commit 07768e03b5a5efc9d768d6afc6246d2ec345cace
Merge: 251cb96 eb07774
Author: Junio C Hamano <gitster@pobox.com>
Date:   Mon Jun 9 11:30:12 2014 -0700

    Merge branch 'jc/shortlog-ref-exclude'
```

It is not necessary to give a full 40 characters of the SHA-1 identifier. Git is smart enough to figure out what you meant if you provide it with the first few characters of the SHA-1 revision identifier, as long as the partial SHA-1 is at least four characters long. To be able to use a

shortened SHA-1 to select revision, it must be long enough to be unambiguous, that is, there is one and only one commit object which SHA-1 identifier begins with given characters.

For example, both `dae86e1950b1277e545cee180551750029cfe735` and `dae86e` name the same commit object, assuming, of course, that there is no other object in your repository whose object name starts with `dae86e`.

In many places, Git shows unambiguous shortened SHA-1 identifiers in its command output. For example, in the preceding example of the `git log` output, we can see the shortened SHA-1 identifiers in the `Merge:` line.

You can also request that Git use the shortened SHA-1 in place of the full SHA-1 revision identifiers with the `--abbrev-commit` option. By default, Git will use at least seven characters for the shortened SHA-1; you can change it with the optional parameter, for example, `--abbrev-commit=12`.

Note that Git would use as many characters as required for the shortened SHA-1 to be unique at the time the command was issued. The parameter `--abbrev-commit` (and the similar `--abbrev` option) is the minimal length.

Tip

HA short note about the shortened SHA-1:

Generally, 8 to 10 characters are more than enough to be unique within a project. One of the largest Git projects, the Linux kernel, is beginning to need 12 characters out of the possible 40 to stay unique. While a hash collision, which means having two revisions (two objects) that have the same full SHA-1 identifier, is extremely unlikely (with $1/2^{80} \approx 1/1.2 \times 10^{24}$ probability), it is possible that formerly unique shortened SHA-1 identifier will stop to be unique due to the repository growth.

The SHA-1 and the shortened SHA-1 are most often copied from the command output and pasted as a revision parameter in another

command. They can also be used to communicate between developers in case of doubt or ambiguity, as SHA-1 identifiers are the same in any clone of the repository. *Fig 2* uses a five-character shortened SHA-1 to identify revisions in the DAG.

Ancestry references

The other main way to specify a revision is via its ancestry. One can specify a commit by starting from some child of it (for example from the current commit i.e. HEAD, a branch head, or a tag), and then follow through parent relationships to the commit in question. There is a special suffix syntax to specify such ancestry paths.

If you place `^` at the end of a revision name, Git resolves it to mean a (first) parent of that revision. For example, `HEAD^` means the parent of the `HEAD`, that is, the previous commit.

This is actually a shortcut syntax. For merge commits, which have more than one parent, you might want to follow any of the parents. To select a parent, put its number after the `^` character; using the `^<n>` suffix means the *n*th parent of a revision. We can see that `^` is actually a short version of `^1`.

As a special case, `^0` means the commit itself; it is important only when a command behaves differently when using the branch name as a parameter and when using other revision specifier. It can be also used to get the commit an annotated (or a signed) tag points to; compare `git show v0.9` and `git show v0.9^0`.

This suffix syntax is composable. You can use `HEAD^^` to mean grandparent of `HEAD`, and parent of `HEAD^`.

There is another shortcut syntax for specifying a chain of first parents. Instead of writing *n* times the `^` suffix, that is, `^...^` or `^1^1...^1`, you can simply use `~<n>`. As a special case, `~` is equivalent to `~1`, so, for example, `HEAD~` and `HEAD^` are equivalent. And, `HEAD~2` means *the first parent of the first parent*, or *the grandparent*, and is equivalent to `HEAD^^`.

You can also combine it all together, for example, you can get the second parent of the great grandparent of `HEAD` (assuming it was a merge commit) by using `HEAD~3^2` and so on. You can use `git name-rev` or `git describe --contains` to find out how a revision is related to local refs, for example, via:

```
$ git log | git name-rev --stdin
```

Reverse ancestry references: the git describe output

The ancestry reference describes how a historic version relates to the current branches and tags. It depends on the position of the starting revision. For example, `HEAD^` would usually mean completely different commit next month.

Sometimes, we want to describe how the current version relates to prior named version. For example, we might want to have a human-readable name of the current version to store in the generated binary application. And, we want this name to refer to the same revision for everybody. This is the task of `git describe`.

The `git describe` finds the most recent tag that is reachable from a given revision (by default, `HEAD`) and uses it to describe that version. If the found tag points to the given commit, then (by default) only the tag is shown. Otherwise, `git describe` suffixes the tag name with the number of additional commits on top of the tagged object, and the abbreviated SHA-1 identifier of the given revision. For example, `v1.0.4-14-g2414721` means that the current commit was based on named (tagged) version `v1.0.4`, which was 14 commits ago, and that it has `2414721` as a shortened SHA-1.

Git understands this output format as a revision specifier.

Reflog shortnames

To help you recover from some of types of mistakes, and to be able to

undo changes (to go back to the state before the change), Git keeps a **reflog**—a *temporary* log of where your `HEAD` and branch references have been for the last few months, and how they got there. The default is to keep reflog entries up to 90 days, 30 days for revisions which are reachable only through reflog (for example, amended commits). This can be, of course, configured, even on a ref-by-ref basis.

You can examine and manipulate your reflog with the `git reflog` command and its subcommands. You can also display `reflog` like a history with `git log -g` (or `git log --walk-reflog`):

```
$ git reflog
ba5807e HEAD@{0}: pull: Merge made by the 'recursive' strategy.
3b16f17 HEAD@{1}: reset: moving to HEAD@{2}
2b953b4 HEAD@{2}: reset: moving to HEAD^
69e0d3d HEAD@{3}: reset: moving to HEAD^^
3b16f17 HEAD@{4}: commit: random.c was too long to type
```

Every time your `HEAD` and your branch head are updated for any reason, Git stores that information for you in this local temporary log of ref history. The data from `reflog` can be used to specify references (and therefore, to specify revisions):

- To specify the *n*th prior value of `HEAD` in your local repository, you can use the `HEAD@{n}` notation that you see in the `git reflog` output. It's same with the *n*th prior value of the given branch, for example, `master@{n}`. The special syntax, `@{n}`, means the *n*th prior value of the current branch, which can be different from `HEAD@{n}`.
- You can also use this syntax to see where a branch was some specific amount of time ago. For instance, to denote where your `master` branch was yesterday in your local repository, you can use `master@{yesterday}`.
- You can use the `@{-n}` syntax to refer to the *n*th branch checked out (used) before the current one. In some places, you can use `-` in place of `@{-1}`, for example, `git checkout -` will switch to the previous branch.

Upstream of remote-tracking branches

The local repository which you use to work on a project does not usually live in the isolation. It interacts with other repositories, usually at least with the `origin` repository it was cloned from. For these remote repositories with which you interact often, Git will track where their branches were at the time of last contact.

To follow the movement of branches in the remote repository, Git uses remote-tracking branches. You cannot create new commits on remote-tracking branches as they would be overwritten on the next contact with remote. If you want to create your own work based on some branch in remote repository, you need to create a local branch based on the respective remote-tracking branch.

For example, when working on a line of development that is to be ultimately published to the `next` branch in the `origin` repository, which is tracked by the remote-tracking branch, `origin/next`, one would create a local `next` branch. We say that `origin/next` is upstream of the `next` branch and we can refer to it as `next@{upstream}`.

The suffix, `@{upstream}` (short form `<refname>@{u}`), which can be applied only to a local branch name, selects the branch that the ref is set to build on top of. A missing ref defaults to the current branch, that is, `@{u}` is upstream for the current branch.

You can find more about remote repositories, the concept of the upstream, and remote tracking branches in [Chapter 5, Collaborative Development with Git](#) and [Chapter 6, Advanced Branching Techniques](#).

Selecting revision by the commit message

You can specify the revision a by matching its commit message with a regular expression. The `:<pattern>` notation (for example, `:^Bugfix`) specifies the youngest matching commit, which is reachable from any ref, while `<rev>^{/<pattern>}` (for example, `next^{/fix bug}`) specifies the youngest matching commit which is reachable from `<rev>`:

```
$ git log 'origin/pu^{/^Merge branch .rs/ref-transaction}'
```

This revision specifier gives similar results to the `--grep=<pattern>` option to `git log`, but is composable. On the other hand, it returns the first (youngest) matching revision, while the `--grep` option returns all matching revisions.

Selecting the revision range

Now that you can specify individual revisions in multiple ways, let's see how to specify ranges of revisions, a subset of the DAG we want to examine. Revision ranges are particularly useful for viewing selected parts of history of a project.

For example, you can use range specifications to answer questions such as, "What work is on this branch that I haven't yet merged into my main branch?" and "What work is on my main branch I haven't yet published?", or simply "What was done on this branch since its creation?".

Single revision as a revision range

History traversing commands such as `git log` operate on a set of commits, walking down a chain of revisions from child to parent. These kind of commands, given a single revision as an argument (as described in the *Single revision selection* section of this chapter), will show the set of commits reachable from that revision, following the commit ancestry chain, all the way down to the root commits.

For example, `git log master` would show all the commits reachable from the tip of a `master` branch (all the revisions that are or were based on the current work on the said branch), which means that it would show the whole `master` branch, the whole line of development.

Double dot notation

The most common range specification is the double-dot syntax, `A..B`. For a linear history, it means all the revisions between `A` and `B`, or to be more exact, all the commits that are in `B` but not in `A`, as shown in *Fig 3*. For example, the range, `HEAD~4..HEAD`, means four commits: `HEAD`, `HEAD^`, `HEAD^^`, and `HEAD^^^` or in other words, `HEAD~0`, `HEAD~1`, `HEAD~2`, and `HEAD~3`, assuming that there are no merge commits starting between the current branch and its fourth ancestor.

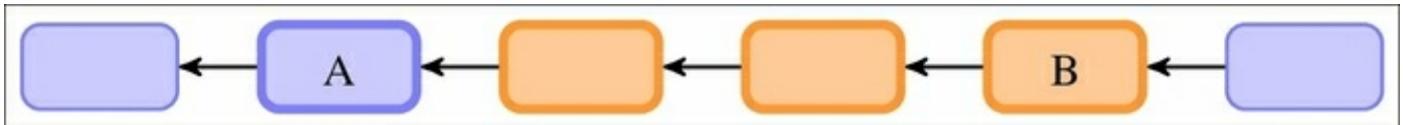


Fig 3. A double dot notation $A..B$ for linear history; the selected revision range is shown in orange

Note

If you want to include a starting commit (in general case: boundary commits), which Git considers uninteresting, you can use the `--boundary` option to `git log`.

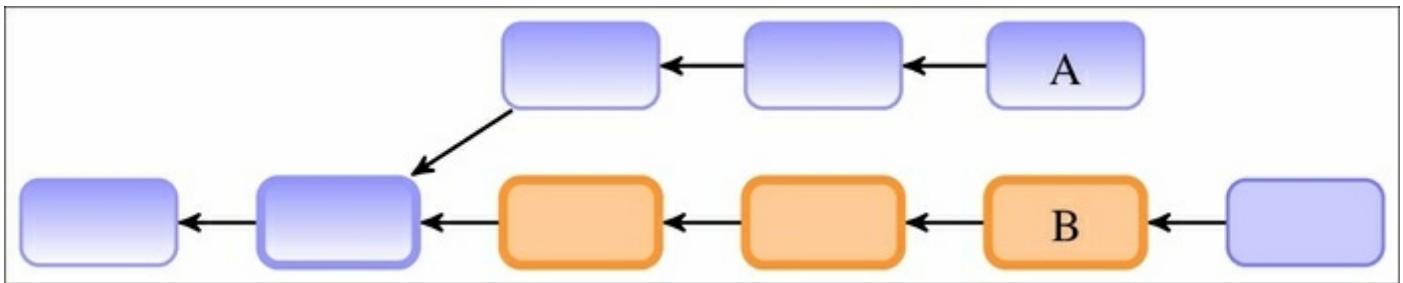


Fig 4. A double dot notation $A..B$ for a non-linear history (revision A is not an ancestor of revision B); where, the selected revision range is orange, while the excluded revisions are shaded, and boundary revision is marked with a thick outline

The situation is more complicated for history that is not a straight line. One such case is when A is not the ancestor of B (there is no path in the DAG of revisions leading from B to A) but both have a common ancestor, like in Fig 4. Another situation with non-linear history is when there are merge commits between A and B , as shown in Fig 5. Precisely in view of nonlinear history the double-dot notation $A..B$, or "between A and B", is defined as reachable from A and not reachable from B .

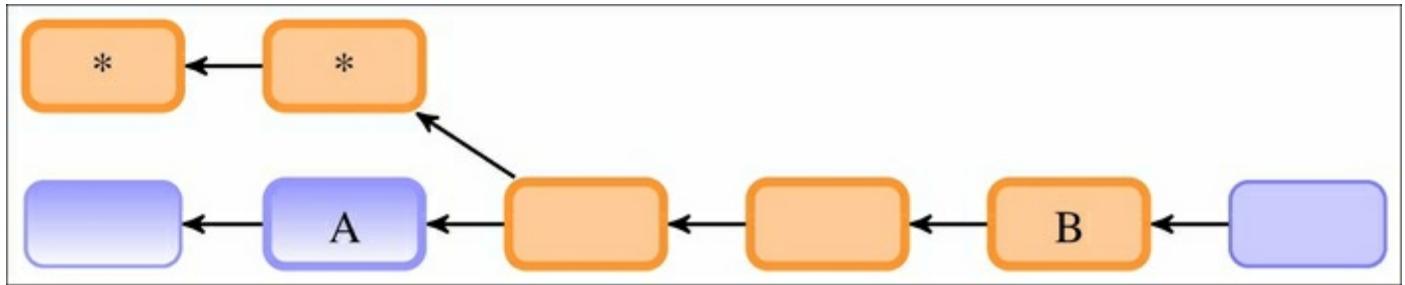


Fig 5 A double dot notation for a non-linear history, with merge commits between A and B. To exclude commits marked with star "" use --strict-ancestor.*

Git `A..B` means a range of all the commits that are reachable from one revision (`B`) but are not reachable from another revision (`A`), following the ancestry chain. In the case of divergent `A` and `B`, like in *Fig 4*, this is simply all commits in `B` from the branch point of `A`.

For example, say your branches `master` and `experiment` diverge. You want to see what is in your `experiment` branch that hasn't yet been merged into your `master` branch. You can ask Git to show you a log of just those commits with `master..experiment`.

If, on the other hand, you want to see the opposite—all the commits in `master` that aren't in `experiment`—you can reverse the branch names. The `experiment..master` notation shows you everything in `master` not reachable from `experiment`.

Note

Another example, `origin/master..HEAD`, shows what you're about to push to the remote (commits in your current branch that are not yet present in the `master` branch in the remote repository `origin`), while `HEAD..origin/master` shows what you have fetched but not yet merged in. You can also leave off one side of the syntax to have Git assume `HEAD: origin/master..` is `origin/master..HEAD` and `..origin/master` is `HEAD..origin/master`; Git substitutes `HEAD` if one side is missing.

Git uses double-dot notation in many places, for example in `git fetch` and `git push` output for an ordinary fast-forward case, so you can just copy and paste a fragment of output as parameters to `git log`. In this case, the beginning of the range is the ancestor of the end of the range; the range is linear:

```
$ git push
To https://git.company.com/random
  8c4ceca..493e222  master -> master
```

Multiple points – including and excluding revisions

The double-dot `A..B` syntax is very useful and quite intuitive, but it is really a shorthand notation. Usually it is enough, but sometimes you might want more than it provides. Perhaps, you want to specify more than two branches to indicate your revision, such as seeing what commits are in any of the several branches that aren't in the branch you're currently on. Perhaps you want to see only those changes on the `master` branch that are not in any of the other long-lived branches.

Git allows you to exclude the commits that are reachable from a given revision by prefixing the said revision with a `^`. For example, to view all the revisions which are `on maint or master` but are not `in next`, you can use `git log maint master ^next`. This means that the `A..B` notation is just a shorthand for `B ^A`.

Instead of having to use `^` character as a prefix for each of the revisions we want to exclude, Git allows you to use the `--not` option, which *negates* all the following revisions. For example, `B ^A ^C` might be written as `B --not A C`. This is useful, for example, when generating those excluded revisions programmatically.

Thus, these three commands are equivalent:

```
$ git log A..B
$ git log B ^A
$ git log B --not A
```

The revision range for a single revision

There is another useful shortcut, namely $A^!$, which is a range composed of a single commit. For non-merge commits, it is simply $A^..A$.

For merge commits, the $A^!$, of course, excludes all the parents. With the help of yet another special notation, namely $A^@$, denoting all the parents of A (A^1, A^2, \dots, A^n), we can say that $A^!$ is a shortcut for $A --not A^@$.

Triple-dot notation

The last major syntax for specifying revision ranges is the triple-dot syntax, $A...B$. It selects all the commits that are reachable by either of the two references, but not by both of them, see *Fig 6*. This notation is called the symmetric difference of A and B .

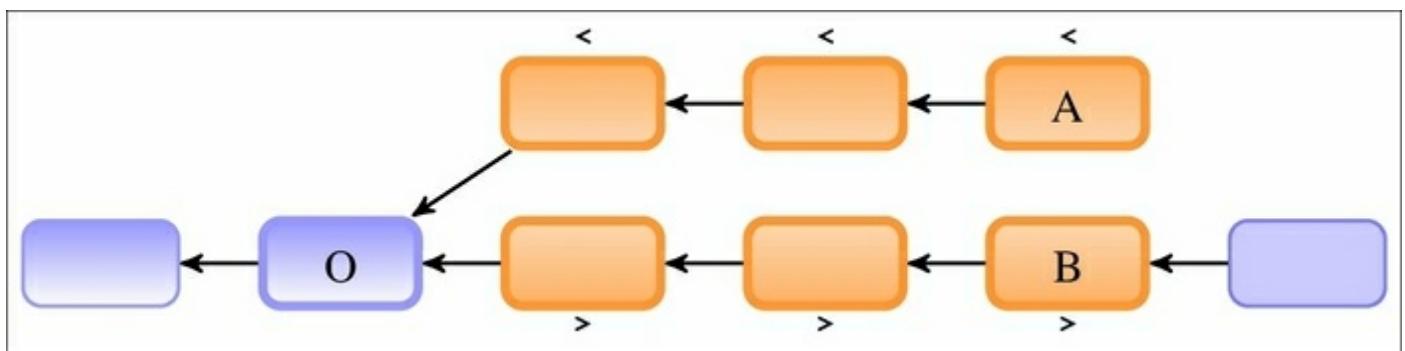


Fig 6. A triple-dot notation $A...B$ for a non-linear history, where the selected range is shown in orange color, boundary commit O is marked with a bold outline, and the characters below and above the nodes show --left-right markers

It is a shortcut notation for $A B --not $(git merge-base --all A B)$, where $\$(...)$ denotes shell **command substitution** (using POSIX shell syntax). Here, it means that the shell will first run the `git merge-base` command to find out all the best common ancestors (all merge bases), and then paste back its output on the command line, to be negated.

A common switch to use with the `git log` command with triple dot notation is `--left-right`. This option makes it show which side of the range each commit is in by prefixing commits from the left side (`A` in `A...B`) with `<`, and those from the right (`B` in `A...B`) with `>`, as shown in *Fig 6* and the following example. This helps make the data more useful:

```
$ git log --oneline --left-right 37ec5ed...8cd8cf8
>8cd8cf8 Merge branch 'fc/remote-helper-refmap' into next
>efcd02e Merge branch 'rs/more-starts-with' into next
>831aa30 Merge branch 'jm/api-strbuf-doc' into next
>1aec19 Merge branch 'jc/revision-dash-count-parsing' into
next
<1a7e8e8 Revert "replace: add --graft option"
<7a30690 t9001: avoid non-portable '\n' with sed
>5cc3268 fetch doc: remove "short-cut" section
```

Note

If the `--left-right` option is combined with `--boundary`, these normally uninteresting boundary commits are prefixed with `-`.

In the case of using a triple-dot `A...B` revision range, these boundary commits are `git merge-base --all A B`.

Git uses triple-dot notation in the `git fetch` and `git push` output when there was a *forced update*, in cases where the old version (left-hand side) and the updated version (right-hand side) diverged, and the new version was forced to overwrite the old version:

```
$ git fetch
From git://git.kernel.org/pub/scm/git/git
 + 37ec5ed...8cd8cf8 next      -> origin/next (forced update)
 + 9478935...16067c9 pu        -> origin/pu (forced update)
   d0b0081..1f58507 todo       -> origin/todo
```

Tip

Using revision range notation in diff

To make it easier to copy and paste revisions between `log` and `diff` commands, Git allows us to use *revision range* double-dot notation `A..B`

and triple-dot `A...B` to denote a *set of revisions (endpoints)* in the `git diff` command.

For Git, using `git diff A..B` is the same as `git diff A B`, which means the difference between revision `A` and revision `B`. If the revision on either side of double dot is omitted, it will have the same effect as using `HEAD` instead. For example, `git diff A..` is equivalent to `git diff A HEAD`.

The `git diff A...B` notation is intended to show the incoming changes on the branch `B`. Incoming changes mean revisions up to `B`, starting at a common ancestor, that is, a merge base of both `A` and `B`. Thus, writing `git diff A...B` is equivalent to `git diff $(git merge-base A B) B`; note that `git merge-base` is without `--all` here. The result of this convention makes it so that a copy and paste of the `git fetch` output (whether with double-dot or triple-dot) as an argument to `git diff` will always show fetched changes. Note, however, that it does not include the changes that were made on `A` since divergence!

Additionally, this feature makes it possible to use a `git diff A^!` command to view how revision `A` differs from its parent (it is a shortcut for `git diff A^ A`).

Searching history

A huge number and variety of useful options to the `git log` command are limiting options—that is, options that let you show only a subset of commits. This complements selecting commits to view by passing the appropriate revision range, and allows us to search the history for the specific versions, utilizing information other than the shape of the DAG of revisions.

Limiting the number of revisions

The most basic way of limiting the `git log` output, the simplest limiting option, is to show only the most recent commits. This is done using the `-<n>` option (where `n` is any integer); this can be also written as `-n <n>`, or in long form as `--max-count=<n>`. For example, `git log -2` would show the two last (most recent) commits in the current line of development, starting from the implicit `HEAD` revision.

You can skip the first few commits shown with `--skip=<n>`.

Matching revision metadata

History limiting options can be divided into those that check information stored in the commit object itself (the revision metadata), and those that filter commits based on changeset (on changes from parent commit(s)).

Time-limiting options

If you are interested in commits created within some date range that you're interested in, you can use a number of options such as `--since` and `--until`, or `--before` and `--after`. For example, the following command gets the list of commits made in the last two weeks:

```
$ git log --since=2.weeks
```

These options work with lots of formats. You can specify a specific date such as "2008-04-21" or a relative date such as "2 years, 3 months, and

3 days ago"; you can use a dot in place of a space.

When using a specific date, one must remember that these dates are interpreted to be in the local time zone, if the date does not include the time zone. It is important because, in such a situation, Git will not yield identical results when run by different colleagues, who may be situated in other time zones around the world. For example, `--since="2014-04-29 12:00:00"` would catch an additional 6 hours, worth of commits when issued in Birmingham, England, United Kingdom (where it means 2014-04-29Z11:00:00 universal time) than when issued in Birmingham, Alabama, USA. (where it means 2014-04-29Z17:00:00). To have everyone get the same results, you need to include the time zone in the time limit, for example, `--after="2013-04-29T17:07:22+0200"`.

Note that Git stores not one but two dates describing the version: author date and committer date. Time-limiting options described here examine the committer date, which means the date and time when the revision object was created. This might be different from author date, which means the date and time when a changeset was created (the change was made).

The date of authorship can be different from the date of committership in a few cases. One is when the commit was created in one repository, converted to e-mail, and then applied by other person in an other repository. Another way to have these two dates differ is to have the commit recreated while rebasing; by default, it keeps the author date and gets a new committer date (refer to [Chapter 8, Keeping History Clean](#)).

Matching commit contents

If you want to filter your commit history to only show those done by a specific author or committer, you can use the `--author` or `--committer` options, respectively. For example, let's say you're looking for all the commits in the Git source code authored by Linus. You could use something like `git log --author=Linus`. The search is, by default, case-sensitive, and uses regular expressions. Git will search both the name

and the e-mail address of the commit author; to match first name only use `--author=^Linus`.

The `--grep` option lets you search commit messages (which should contain descriptions of the changes). Let's say that you want to find all the security bug fixes that mention the Common Vulnerabilities and Exposures (CVE) identifier in the commit message. You could generate a list of such commits with `git log --grep=CVE`.

If you specify both `--author` and `--grep` options, or more than one `--author` or `--grep` option, Git will show commits that match either query. In other words, Git will logically OR all the commit matching options. If you want to find commits that match all the queries, with matching options logically AND, you need to use the `--all-match` option.

There is also a set of options to modify the meaning of matching patterns, similar to the ones used by the `grep` program. To make the search case-insensitive, use the `-i / --regexp-ignore-case` option. If you want to match simply a substring, you can use `-F / --fixed-strings` (you might want to do it to avoid having to escape regular expression metacharacters such as `".` and `"?"`). To write more powerful search terms, you can use `--extended-regexp` or `--perl-regexp` (use the last one only if Git was compiled linked with the PCRE library).

Commit parents

Git, by default, will follow all the parents of each merge commit, when walking down the ancestry chain. To make it follow only the first parent, you can use the aptly named `--first-parent` option. This will show you the main line of the history (sometimes called the trunk), assuming that you follow the specific practices with respect to merging changes; you will learn more about this in [Chapter 7, Merging Changes Together](#).

Compare (this example uses the very nice `--graph` option that makes an ASCII-art diagram of the history) the following code...

```
$ git log -5 --graph --oneline
* 50f84e3 Update draft release notes to 2.1
```

```
* 07768e0 Merge branch 'jc/shortlog-ref-exclude'  
| \  
| * eb07774 shortlog: allow --exclude=<glob> to be passed  
* | 251cb96 Merge branch 'mn/sideband-no-ansi'  
| \ \  
| * | 38de156 sideband.c: do not use ANSI control sequence
```

...with this:

```
$ git log -5 --graph --oneline --first-parent  
* 50f84e3 Update draft release notes to 2.1  
* 07768e0 Merge branch 'jc/shortlog-ref-exclude'  
* 251cb96 Merge branch 'mn/sideband-no-ansi'  
* d37e8c5 Merge branch 'rs/mailinfo-header-cmp'  
* 53b4d83 Merge branch 'pb/trim-trailing-spaces'
```

You can filter the list to show only the merge commits, or show only the non-merge commits, with the `--merges` and `--no-merges` options, respectively. These options can be considered just a shortcut for a more generic options: `--min-parents=<number>` (`--merges` is `--min-parents=2`) and `--max-parents=<number>` (`--no-merges` is `--max-parents=1`).

Let's say that you want to find the starting point(s) of your project. You can do this with the help of `--max-parents=0`, which would give you all the root commits:

```
$ git log --max-parents=0 --oneline  
0ca71b3 basic options parsing and whatnot.  
16d6b8a Initial import of a python script...  
cb07fc2 git-gui: Initial revision.  
161332a first working version  
1db95b0 Add initial version of gitk to the CVS repository  
2744b23 Start of early patch applicator tools for git.  
e83c516 Initial revision of "git", the information manager from hell
```

Searching changes in revisions

Sometimes, searching through commit messages and other revision metadata is not enough. Perhaps, descriptions of the changes were not detailed enough. Or, what if you are looking for a revision when a

function was introduced, or where variables started to be used?

Git allows you to look through the changes that each revision brought (the difference between commit and its parent). The faster option is called a `pickaxe` search.

With the `-S<string>` option, Git will look for differences that introduce or remove an instance of a given string. Note that this is different from the string simply appearing in diff output. (You can do a match using a regular expression with the `--pickaxe-regex` option.) Git checks for each revision if there are files whose *current* side and whose *parent* side have a different number of specified strings, and shows the revisions that match.

As a side effect, `git log` with the `-S` option would also show the changes that each revision made (as if the `--patch` option were used), but only those differences that match the query. To show differences for all the files, and not only those diffs where the change in number occurred, you need to use the `--pickaxe-all` option:

```
$ git log -S'sub href'  
commit 06a9d86b49b826562e2b12b5c7e831e20b8f7dce  
Author: Martin Waitz <tali@admingilde.org>  
Date:   Wed Aug 16 00:23:50 2006 +0200
```

```
    gitweb: provide function to format the URL for an action  
link.
```

```
Provide a new function which can be used to generate an URL  
for the CGI.
```

```
This makes it possible to consolidate the URL generation in  
order to make
```

```
it easier to change the encoding of actions into URLs.
```

```
Signed-off-by: Martin Waitz <tali@admingilde.org>  
Signed-off-by: Junio C Hamano <junkio@cox.net>
```

With `-G<regex>`, Git will literally look for differences whose added or removed line matches the given regular expression. Note that the unified diff format (that Git uses) considers changing line as removing the old

version and adding a new one; refer to [Chapter 3, Developing with Git](#) for an explanation of how Git describes changes.

To illustrate the difference between `-S<regex> --pickaxe-regex` and `-G<regex>`, consider a commit with the following `diff` in the same file:

```
if (lstat(path, &st))
-
return error("cannot stat '%s': %s", path,
+
ret = error("cannot stat '%s': %s", path,
                strerror(errno));
```

While `git log -G"error\("` will show this commit (because the query matches both changed lines), `git log -S"error\("` --pickaxe-regex will not (because the number of occurrences of that string did not change).

Note

If we are interested in a single file, it is easier to use `git blame` (perhaps in a graphical blame browser, like with `git gui blame`) to check when the given change was introduced. However, `git blame` can't be used to find a commit that deleted a line—you need a pickaxe search for that.

Selecting types of change

Sometimes, you might want to see only those changes that added or renamed files. With Git, you can do this with `git log --diff-filter=AM`. You can select any combination of types of changes; see the `git-log(1)` manpage for details.

History of a file

As described in the *Whole-tree commits* section at the beginning of this chapter, in Git revisions are about the state of the whole project as one single entity.

In many cases, especially with larger projects, we are interested only in the history of a single file, or in the history limited to the changes in the given directory (in the given subsystem).

Path limiting

To examine the history of a single file, you can simply use `git log <pathname>`. Git will then only show all those revisions that affected the pathname (a file or a directory) given, which means those revisions where there was a change to the given file, or a change to a file inside the given subdirectory.

Tip

Disambiguation between branch names and path names

Git usually guesses what you meant by writing `git log foo`; did you meant to ask for the history of branch `foo` (line of development), or for the history of the file `foo`. However, sometimes Git can get confused. To prevent confusion between filenames and branch names, you can use `--` to separate filename arguments from other options. Everything after `--` will be taken to be a pathname, everything before it will be taken to be the branch name or other option.

For example, writing `git log -- foo` explicitly asks for the history of a path `foo`.

One of the common situations where it is needed, besides having the same name for a branch and for a file, is examining the history of a deleted file, which is no longer present in a project.

You can specify more than one path; you can even look for the changes that affect the given type of file with the help of wildcards (pattern matching). For example, to find only changes to Perl scripts (to files with the `*.pl` extension), you can use `git log -- '*.pl'`. Note that you need to protect the `*.pl` wildcard from being expanded by the shell, before Git sees it, for example via single quotes as shown here.

However, as Git uses pathname parameters as limiters in showing the history of a project, querying for a history of a single file doesn't automatically *follow renames*. You need to use `git log --follow <file>` to continue listing the history of a file beyond renames. Unfortunately, it doesn't work in all the cases. Sometimes, you need to use either the blame command (see the next section), or examine boundary commits with rename detection turned on (`git show -M -C --raw --abbrev <rev>`) and follow renames and file moving *manually*.

In modern Git, you can also trace the evolution of the line range within the file using `git log -L`, which is currently limited to walk starting from a single revision (zero or one positive revision arguments) and a single file. The range is given either by denoting the start and end of the range with `-L <start>,<end>:<file>` (where either `<start>` or `<end>` can be the line number or `/regexp/`), or a function to track with `-L :<funcname regexp>:<file>`. This cannot be used together with the ordinary spec-based path limiting.

History simplification

By default, when requested for the history of a path, Git would simplify the history, showing only those commits that are required (that are enough) to explain how the files that match the specified paths came to be. Git would exclude those revisions that do not change the given file. Additionally, for non-excluded merge commits, Git would exclude those parents that do not change the file (thereby excluding lines of development).

You can control this kind of history simplification with the `git log` options such as `--full-history` or `--simplify-merges`. Check the Git

documentation for more details, like the "History Simplification" section in `git-log(1)` manpage.

Blame – the line-wise history of a file

The `blame` command is a version control feature designed to help you determine who made changes to a file. This command shows for each line in the file when this line was created, who authored given line, and so on. It does that by finding the latest commit in which the current shape of each line was introduced. A revision introducing given shape is the one where the given line has its current form, but where the line is different in this revision parent. The default output of `git blame` annotates each line with appropriate line-authorship information.

Git can start annotating from the given revision (useful when browsing the history of a file or examining how older version of a file came to be) or even limit the search to a given revision range. You can also limit the range of lines annotated to make blame faster—for example to check only the history of an `esc_html` function in `gitweb/gitweb.perl` file you can use:

```
$ git blame -L '/^sub esc_html {/,/}' gitweb/gitweb.perl
```

What makes blame so useful is that it follows the history of file across whole-file renames. It can optionally follow lines as they were moved from one file to another (with the `-M` option), and even follow lines that were copied and pasted from another file (with the `-C` option); this includes internal code movement.

When following code movement, it is useful to ignore changes in whitespace, to find out when given fragment of code was truly introduced and avoid finding when it was just re-indented (for example, due to refactoring repeated code into a function—code movement). This can be done by passing the `diff` formatting option `-w` or `--ignore-all-space`.

Tip

Rename detection

Good version control systems should be able to deal with renaming files and other ways of changing the directory structure of a project. There are two ways to deal with this problem. The first is the **rename tracking**, which means that the information about the fact that a file was renamed is saved at the commit time; the version control system marks renames. This usually requires using the `rename` and `move` commands to rename files (no use of non-version control aware file managers), or it can be done by detecting the rename at the time of creating the revision. It can involve some kind of **file identity** surviving across renames.

The second method, and the one used by Git, is the **rename detection**. In this case, the `mv` command is only a shortcut for deleting a file with the old name and adding a file with the same contents and a new name. Rename detection means that the fact that file was renamed is detected at the time it is needed: when doing a merge, viewing the line-wise history of a file (if requested), or showing a diff (if requested or configured). This has the advantage that the rename detection algorithm can be improved, and is not frozen at the time of commit. It is a more generic solution, allowing to handle not only the whole-file renames, but also the code movement and copying within a single file and across different files, as can be seen in the description of `git blame`.

The disadvantage of the rename detection, which in Git is based on the heuristic similarity of the file contents and pathname, is that it takes resources to run, and that in rare cases it can fail, not detecting renames or detecting a rename where there isn't one.

Note that, in Git, rename detection is not turned on for diffs by default.

Many graphical interfaces for Git include a graphical version of blame. The `git gui blame` command is an example of such a graphical interface to blame operation (it is a part of `git gui`, a Tcl/Tk-based graphical interface). Such graphical interfaces can show the full description of changes and simultaneously show the history with and

without considering renames. From such a GUI, it is usually possible to go to a specified commit, browsing the history of lines of a file interactively. In addition, the GUI blame tool makes it very easy to follow files across renames.

The screenshot shows a window titled "File Viewer" with the following details:

- Commit:** master
- File:** fetch-pack.c
- Code Snippet:** A portion of C code from line 51 to 56, showing the detection of copied or moved code. The code includes:

```
51 static unsigned int allow_unadvertised_object_
52
53 static void rev_list_push(struct commit *commi
54 {
55     if (!(commit->object.flags & mark)) {
56         commit->object.flags |= mark;
```
- Annotations:**
 - Originally By:** 745f 006: commit 23d61f8343282643c830e1c446962b213dbcc09a NTND JI Johannes Schindelin Fri Oct 28 04:46:27 2005
 - Subject:** [PATCH] git-fetch-pack: Do not use git-rev-list
 - Copied Or Moved Here By:** 745f 745: commit 745f7a8cacae55df3e00507344d8db2a31eb57e8 NTND NTNI Nguyen Thai Ngoc Duy Fri Oct 26 17:53:55 2012
 - Subject:** [PATCH] git-fetch-pack: move core code to libgit.a
- Commit Log:**

```
commit 23d61f8343282643c830e1c446962b213dbcc09a
Author: Johannes Schindelin <Johannes.Schindelin@gmx.de> Fri Oct 28 04:46:27 2005
Committer: Junio C Hamano <gitster@pobox.com> Sat Oct 29 07:56:58 2005

Subject: [PATCH] git-fetch-pack: Do not use git-rev-list
```
- Status Bar:** Annotation complete.

Fig 7. The GUI blame in action, showing the detection of copying or moving fragments of code

Finding bugs with git bisect

Git provides a couple of tools to help you debug issues in your projects. These tools can be extremely useful, especially in the case of a software regression, a software bug which makes a feature stop functioning as intended after a certain revision. If you don't know where the bug is, and there have been dozens or hundreds of commits since the last state where you know the code worked, you'll likely turn to `git bisect` for help.

The `bisect` command searches semi-automatically step by step through the project history, trying to find the revision that introduced the bug. In each step, it bisects the history into roughly equal parts, and asks whether there is a bug in the dividing commit. It then uses the answer to eliminate one of the two sections, and reduces the size of the revision range where there can be a commit that introduced the bug.

Suppose version 1.14 of your project worked, but the release candidate, 1.15-rc0, for the new version crashes. You go back to the 1.15-rc0 version, and it turns out that you can *reproduce the issue* (this is very important!), but you can't figure out what is going wrong.

You can bisect the code history to find out. You need to start the bisection process with `git bisect start`, and then tell Git which version is broken with `git bisect bad`. Then, you must tell bisect the last-known good state (or set of states) with `git bisect good`:

```
$ git bisect start
$ git bisect bad v1.15-rc0
$ git bisect good v1.14
Bisecting: 159 revisions left to test after this (roughly 7
steps)
[7ea60c15cc98ab586aea77c256934acd438c7f95] Merge branch
'mergetool'
```

Git figured out that about 300 commits came between the commit you marked as the last good commit (`v1.14`) and the bad version (`v1.15-rc0`), and it checked out the middle one (`7ea60c15`) for you. If you run

git branch at this point, you'll see that git has temporarily moved you to (no branch):

```
$ git branch
* (no branch, bisect started on master)
  master
```

At this point, you need to run your test to check whether the issue is present in the commit currently checked out by the bisect operation. If the program crashes, mark this commit as bad with `git bisect bad`. If the issue is not present, mark it as correct with `git bisect good`. After about seven steps, Git would show the suspect commit:

```
$ git bisect good
b047b02ea83310a70fd603dc8cd7a6cd13d15c04 is first bad commit
commit b047b02ea83310a70fd603dc8cd7a6cd13d15c04
Author: PJ Hyett <pjhyett@example.com>
Date:   Tue Jan 27 14:48:32 2009 -0800

  secure this thing

:040000 040000 40ee3e7... f24d3c6... M config
```

The last line in the preceding example output is in so called *raw* diff output, showing which files changed in a commit. You can then examine the suspected commit with `git show`. From there, you can find the author of the said commit, and ask them for clarification, or to fix it (or to send them a bug report). If the good practice of creating small incremental changes was followed during the development of the project, the amount of code to examine after finding the bad commit should be small.

If at any point, you land on a commit that broke something unrelated, and is not a good one to test, you can skip such a commit with `git bisect skip`. You can even skip a range of commits by giving the revision range to the `skip` subcommand.

When you're finished, you should run `git bisect reset` to return you to the branch you started from:

```
$ git bisect reset
Previous HEAD position was b047b02... secure this thing
Switched to branch 'master'
```

To finish bisection while staying on located bad commit, you can use `git bisect reset HEAD`.

You can even fully automate finding bad revision with '`git bisect run`'. For this, you need to have a script that will test for the presence of bug, and exit the value of `0` if the project works all right, or non-`0` if there is a bug. The special exit code, `125`, should be used when the currently checked out code cannot be tested. First, you again tell it the scope of the `bisect` by providing the known bad and good commits. You can do this by listing them with the `bisect start` command if you want, listing the known bad commit first and the known good commit(s) second. You can even cut down the number of trials, if you know what part of the tree is involved in the problem you are tracking down, by specifying path parameters:

```
$ git bisect start v1.5-rc0 v1.4 -- arch/i386
$ git bisect run ./test-error.sh
```

Doing so automatically runs `test-error.sh` on each checked-out commit until Git finds the first broken commit. Here, we have provided the scope of the `bisect` by putting known bad and good commits with the `bisect start` command, listing the known bad commit first and the known good commit(s) second.

If the bug is that the project stopped compiling (a broken build), you can use `make` as a test script (`git bisect run make`).

Selecting and formatting the git log output

Now that you know how to select revisions to examine and to limit which revisions are shown (selecting those that are interesting), it is time to see how to select which part of information associated with the queried revisions to show, and how to format this output. There is a huge number and variety of options of the `git log` command available for this.

Predefined and user defined output formats

A very useful `git log` option is `--pretty`. This option changes the format of log output. There are a few prebuilt formats available for you to use. The `oneline` format prints each commit on a single line, which is useful if you're looking at a lot of commits; there exists `--oneline`, shorthand for `--pretty=oneline --abbrev-commit` used together. In addition, the `short`, `medium` (the default), `full`, and `fuller` formats show the output in roughly the same format, but with less or more information, respectively. The `raw` format shows commits in the internal Git representation. It is possible to change the format of dates shown in those verbose pretty formats with an appropriate `--date` option: make Git show relative dates, like for example *2 hours ago*, with `--date=relative`, dates in your local time zone with `--date=local`, and so on.

You can also specify your own log format with `--pretty=format:"<string>"` (and its `tformat` variant). This is especially useful when you're generating output for machine parsing, for use in scripts, because when you specify the format explicitly you know it won't change with updates to Git. The format string works a little bit like in `printf`:

```
$ git log --pretty="%h - %an, %ar : %s"
50f84e3 - Junio C Hamano, 7 days ago : Update draft release
notes
0953113 - Junio C Hamano, 10 days ago : Second batch for 2.1
```

afa53fe - Nick Alcock, 2 weeks ago : t5538: move http push tests out

There is a very large number of placeholders selected of those are listed in the following table:

Placeholder	Description of output
%H	Commit hash (full SHA-1 identifier of revision)
%h	Abbreviated commit hash
%an	Author name
%ae	Author e-mail
%ar	Author date, relative
%cn	Committer name
%ce	Committer email
%cr	Committer date, relative
%s	Subject (first line of a commit message, describing revision)
%%	A raw %

Tip

Author versus committer

The *author* is the person who originally wrote the patch (authored the changes), whereas the *committer* is the person who last applied the patch (created a commit object with those changes, representing the revision in the DAG). So, if you send a patch to a project and one of the core members applies the patch, both of you get credit—you as the author and the core member as the committer.

The `--oneline` format option is especially useful together with another `git log` option called `--graph`; though it can be used with any format. The latter option adds a nice little ASCII graph showing your branch and merge history. To see where tags and branches are, you can use an option named `--decorate`:

```
$ git log --graph --decorate --oneline origin/maint
*   bce14aa (origin/maint) Sync with 1.9.4
  \
  | * 34d5217 (tag: v1.9.4) Git 1.9.4
  | * 12188a8 Merge branch 'rh/prompt' into maint
  | \
  | * \   64d8c31 Merge branch 'mw/symlinks' into maint
  | |\ \
* | | | d717282 t5537: re-drop http tests
* | | | e156455 (tag: v2.0.0) Git 2.0
```

You might want to use a graphical tool to visualize your commit history. One such tool is a Tcl/Tk program called `gitk` that is distributed with Git. You can find more information about various types of graphical tools in [Chapter 10, *Customizing and Extending Git*](#).

Including, formatting, and summing up changes

You can examine single revision with the `git show` command, which, in addition to the commit metadata, shows changes in the unified `diff` format. Sometimes, however, you might want to display changes alongside the selected part of the history in the `git log` output. You can do this with the help of the `-p` option. This is very helpful for code review, or to quickly browse what happened during a series of commits that a collaborator has added.

Ordinarily, Git would not show the changes for a merge commit. To show changes from all parents, you need to use the `-c` option (or `-cc` for compressed output), and to show changes from each parent individually, use `-m`.

The `git log` accepts various options to change the format of diff output.

Sometimes, it's easier to review changes on the word level rather than on the line level. The `git log` command accepts various options to change the format of diff output. One of those options is `--word-diff`. This way of viewing differences is useful for examining changes in documents (for example, documentation):

```
commit 06ab60c06606613f238f3154cb27cb22d9723967
Author: Jason St. John <jstjohn@purdue.edu>
Date:   Wed May 21 14:52:26 2014 -0400
```

Documentation: use "command-line" when used as a compound adjective, and fix

```
Signed-off-by: Jason St. John <jstjohn@purdue.edu>
Signed-off-by: Junio C Hamano <gitster@pobox.com>
```

```
diff --git a/Documentation/config.txt
b/Documentation/config.txt
index 1932e9b..553b300 100644
--- a/Documentation/config.txt
+++ b/Documentation/config.txt
@@ -381,7 +381,7
    Set the path to the root of the working tree.
    This can be overridden by the GIT_WORK_TREE environment
    variable and the '--work-tree' [-command line-]
{+command-line+} option.
    The value can be an absolute path or relative to the
path to
        the .git directory, which is either specified by --git-
dir
        or GIT_DIR, or automatically discovered.
```

Another useful set of options are about ignoring changes in whitespace, including `-w` / `--ignore-all-space` to ignore all whitespace changes, and `-b` / `--ignore-space-change` to ignore changes in the amount of whitespace.

Sometimes, you are interested only in the summary of changes, and not the details. There is a series of `diff` summarizing options that you can use. If you want to know only which files changed, use `--names-only` (or `--raw --abbrev`). If you also want to know how much those files changed, you can use the `--stat` option (or perhaps its machine-parse

friendly version, `--numstat`) to see some abbreviated stats. If you are interested only in short summary of changes, use `--shortstat` or `--summary`.

Summarizing contributions

Ever wondered how many commits you've contributed to a project? Or perhaps, who is the most active developer during the last month (with respect to the number of commits)? Well, wonder no more, because this is what `git shortlog` is good for:

```
$ git shortlog -s -n
13885 Junio C Hamano
1399 Shawn O. Pearce
1384 Jeff King
1108 Linus Torvalds
 743 Jonathan Nieder
```

The `-s` option squashes all of the commit messages into the number of commits; without it, `git shortlog` would list summary of all the commits, grouped by developer (its output can be configured to some extent with `pretty` like the `--format` option.) The `-n` option sorts the list of developers by the number of commits; otherwise, it is sorted alphabetically. You can add an `-e` option to show also an e-mail address; note that, however, with this option, Git will separate contributions made by the same author under different e-mail.

The `git shortlog` command accepts a revision range, and other revision limiting options such as `--since=1.month.ago`; almost options that `git log` command accepts makes sense for `shortlog`. For example, to see who contributed to the last release candidate you can use the following command:

```
$ git shortlog -e v2.0.0-rc2..v2.0.0-rc3
Jonathan Nieder <jrnieder@gmail.com> (1):
    shell doc: remove stray "+" in example

Junio C Hamano <gitster@pobox.com> (14):
    Merge branch 'cl/p4-use-diff-tree'
    Update draft release notes for 2.0
```

```
Merge branch 'km/avoid-cp-a' into maint
```

Note

One needs to remember that the number of revisions authored is only one way of measuring contribution. For example, somebody, who creates buggy commits only to fix them later, would have a larger number of commits than the developer who doesn't make mistakes (or cleans the history before publishing changes).

There are other measures of programmer productivity, for example, the number of changed lines in authored commits, or the number of *surviving* lines—these can be calculated with the help of Git, but there is no built-in command to calculate them.

Tip

Mapping authors

One problem with running `git shortlog -s -n -e` or `git blame` in Git repositories of long running projects is that authors may change their name or e-mail, or both during the course of the project, due to many reasons: changing work (and work e-mail), misconfiguration, spelling mistakes, and others:

```
Bob Hacker <bob@example.com>
Bob <bob@example.com>
```

When that happens, you can't get proper attribution. Git allows you to coalesce author/e-mail pairs with the help of the `.mailmap` file in the top directory of your project. It allows us to specify *canonical* names for contributors, for example:

```
Bob Hacker <bob@example.com>
```

(Actually it allows us to specify the canonical name, canonical e-mail, or both name and email, matching by email or name and email.)

By default, those corrections are applied to `git blame` and to `git shortlog`, but not to the `git log` output. With custom output, you can,

however, use placeholders that output corrected name, or corrected e-mail; or you can use the `--use-mailmap` option, or the `log.mailmap` configuration variable.

Viewing a revision and a file at revision

Sometimes, you might want to examine a single revision (for example, a commit suspected to be buggy, found with `git bisect`) in more detail, examining together changes with their description. Or perhaps, you want to examine the tag message of an annotated tag together with the commit it points to. Git provides a generic `git show` command for this; it can be used for any type of object.

For example, to examine the grandparent of the current version, use the following command:

```
$ git show HEAD^^  
commit ca3cdd6bb3fc0c162a690d5383bdb8e8144b0d2  
Author: Bob Hacker <bob@virtech.com>  
Date:   Sun Jun 1 02:36:32 2014 +0200
```

Added COPYRIGHT

```
diff --git a/COPYRIGHT b/COPYRIGHT  
new file mode 100644  
index 000000..862aaaf  
--- /dev/null  
+++ b/COPYRIGHT  
@@ -0,0 +1,2 @@  
+Copyright (c) 2014 VirTech Inc.  
+All Rights Reserved
```

The `git show` command can also be used to display directories (trees) and file contents (blobs). To view a file (or a directory), you need to specify where it is from (from which revision) and the path to the file, using `:` to connect them. For example, to view the contents of the `src/rand.c` file as it was in the version tagged `v0.1` use:

```
$ git show v0.1:src/rand.c
```

This might be more convenient than checking out the required version of

the file into the working directory with `git checkout v0.1 -- src/rand.c`. Before the colon may be anything that names a commit (`v0.1` here), and after that, it may be any path to a file tracked by Git (`src/rand.c` here). The pathname here is the full path from the top of the project directory, but you can use `./` after the colon for relative paths, for example, `v0.1:./rand.c` if you are in the `src/` subdirectory.

You can use the same trick to compare arbitrary files at arbitrary revisions.

Summary

This chapter showed us the various ways of exploring project history: finding relevant revisions, selecting and filtering revisions to display, and formatting the output.

We started with the description of the conceptual model of project history: the Directed Acyclic Graph (DAG) of revisions. Understanding this concept is very important because many selection tools refer directly or indirectly to the DAG.

Then, you learnt how to select a single revision and the range of revisions. We can use this knowledge to see what changes were made on a branch since its divergence from the base branch, and to find all the revisions which were made by the given developer.

We can even try to find bugs in the code by exploring the history: finding when a function was deleted from the code with a `pickaxe` search, examining a file for how its code came to be and who wrote it with `git blame`, and utilizing semi-automatic or automatic searches through the project history to find which version introduced regression with `git bisect`.

When examining a revision, we can select the format in which the information is shown, even to the point of user-defined formats. There are various ways of summarizing the information, from the statistics of the changed files to the statistics of the number of commits per author.

Chapter 3. Developing with Git

The previous chapter explained how to examine the project history. This chapter will describe how to create such history and how to add to it. We will learn how to create new revisions and new lines of development. Now it's time to show how to develop with Git.

Here we will focus on committing one's own work, on the solo development. The description of working as one of the contributors is left for [Chapter 5, Collaborative Development with Git](#), while [Chapter 7, Merging Changes Together](#), shows how Git can help in maintainer duties.

This chapter will introduce the very important Git concept of the staging area (the index). It will also explain, in more detail, the idea of a detached HEAD, that is, an anonymous unnamed branch. Here you can also find a detailed description of the extended unified diff format that Git uses to describe changes.

The following is the list of the topics we will cover in this chapter:

- The index – a staging area for commits
- Examining the status of the working area and changes in it
- How to read the extended unified diff that is used to describe changes
- Selective and interactive commit, and amending a commit
- Creating, listing, and selecting (switching to) branches
- What can prevent switching branch, and what you can do then
- Rewinding a branch with `git reset`
- Detached HEAD, that is, the unnamed branch (checking out tag and so on)

Creating a new commit

Before starting to develop with Git, you should introduce yourself with a name and an e-mail, as shown in [Chapter 1, Git Basics in Practice](#). This

information will be used to identify your work, either as an author or as a committer. The setup can be global for all your repositories (with `git config --global`, or by editing the `~/.gitconfig` file directly), or local to a repository (with `git config`, or by editing `.git/config`). The per-repository configuration overrides the per-user one (you will learn more about it in [Chapter 10, Customizing and Extending Git](#)). You might want to use your company e-mail for work repositories, but your own non-work e-mail for public repositories you work on.

A relevant fragment of the appropriate `config` file could look similar to this:

```
[user]
name = Joe R. Hacker
email = joe@company.com
```

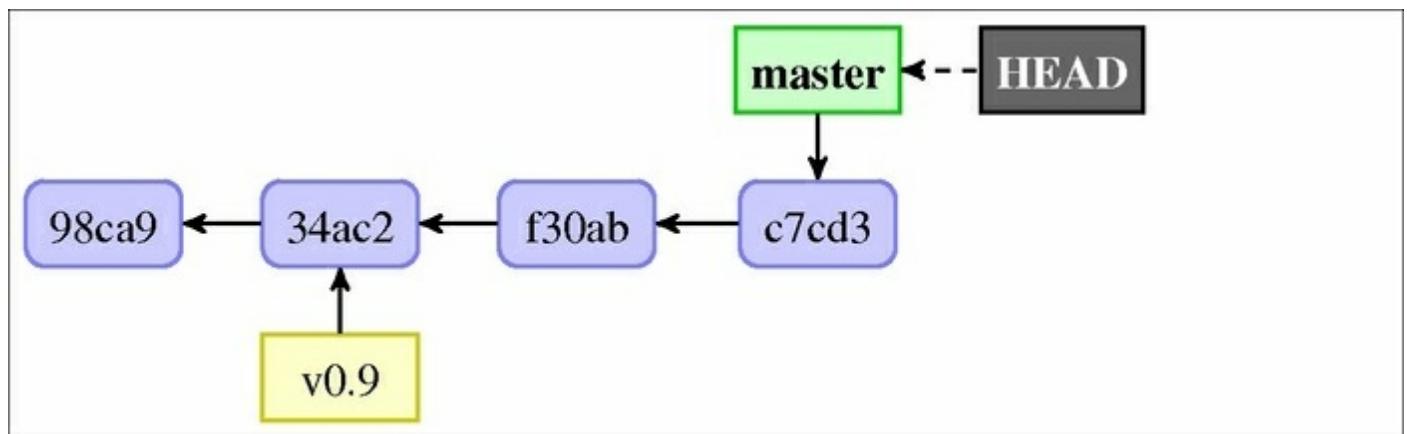


Fig 1. The graph of revisions (the DAG) for a starting point of an example project, before creating a new commit. The current branch is master, and its tip is at revision c7cd3; this is also currently checked out revision, which can be referred to as HEAD.

The DAG view of creating a new commit

[Chapter 2, Exploring Project History](#), introduced the concept of *Directed Acyclic Graph* (DAG) of revisions. Contributing to the development of a project usually consists of creating new revisions of the said project, and adding them as commit nodes to the graph of

revisions.

Let's assume that we are on the `master` branch, as shown in *Fig 1* of the preceding section, and that we want to create a new version (the details of this operation will be described in more detail later). The `git commit` command will create a new commit object—a new revision node. This commit will have as a parent the checked out revision (`c7cd3` in the example). That revision is found by following refs starting from `HEAD`; here, it is `HEAD` to `master` to `c7cd3` chain.

Then Git will move the parent pointer to the new node, creating a situation as in *Fig 2*. In it, the new commit is marked with a thick red outline, and the old position of the `master` branch is shown semi-transparent. Note that the `HEAD` pointer doesn't change; all the time it points to `master`:

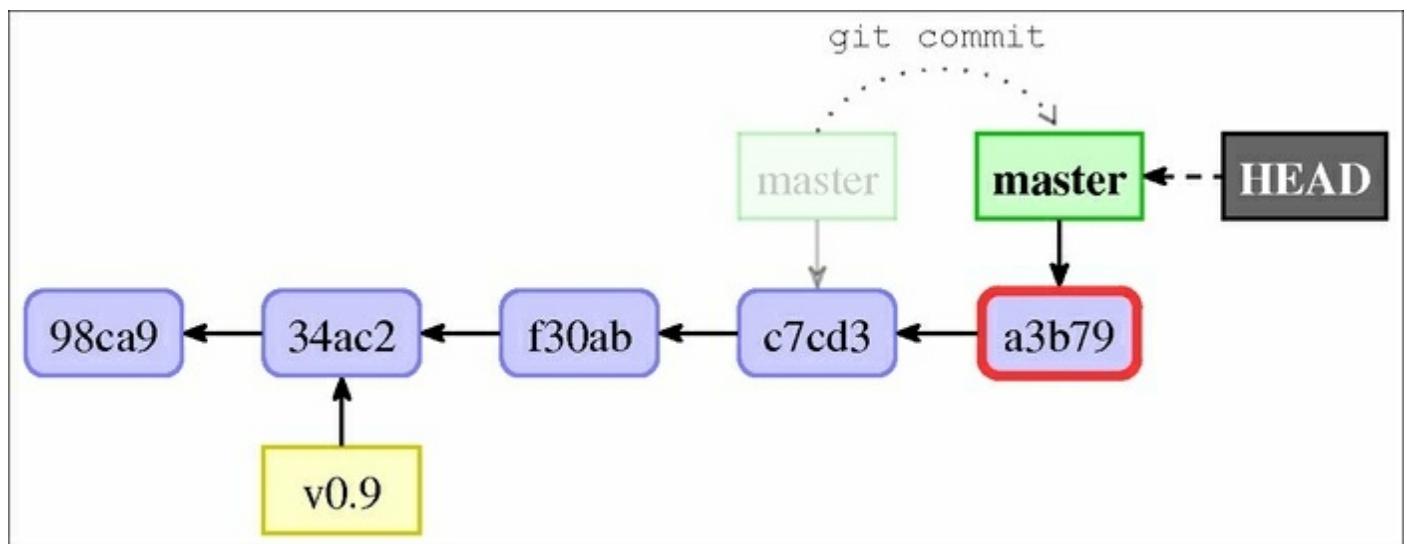


Fig 2: The graph of revisions (the DAG) for an example project just after creating a new commit, starting from the state given by Fig 1

The new commit, `a3b79`, is marked with the thick red outline. The tip of the `master` branch changes from pointing to commit `c7cd3` to pointing to commit `a3b79`, as shown with the dotted line.

The index – a staging area for commits

Each of your files inside the working area of the Git repository can be either known or unknown to Git (be a **tracked file**). The files unknown to Git can be either untracked or ignored (you can find more information about ignoring files in [Chapter 4, Managing Your Worktree](#)).

Files tracked by Git are usually in either of the two states: committed (or unchanged) or modified. The committed state means that the file contents in the working directory is the same as in the last release, which is safely stored in the repository. The file is modified if it has changed compared to the last committed version.

But, in Git, there is another state. Let's consider what happens when we use the `git add` command to add a file, but did not yet create a new commit adding it. A version control system needs to store such information somewhere. Git uses something called the **index** for this; it is the *staging area* that stores information that will go into the next commit. The `git add <file>` command *stages* the current contents (current version) of the file, adding it to the index.

Note

If you want to only mark a file for addition, you can use `git add -N <file>`; this stages empty contents for a file.

The index is a third section storing copy of a project, after a working directory (which contains your own copy of the project files, used as a private isolated workspace to make changes), and a local repository (which stores your own copy of a project history, and is used to synchronize changes with other developers):

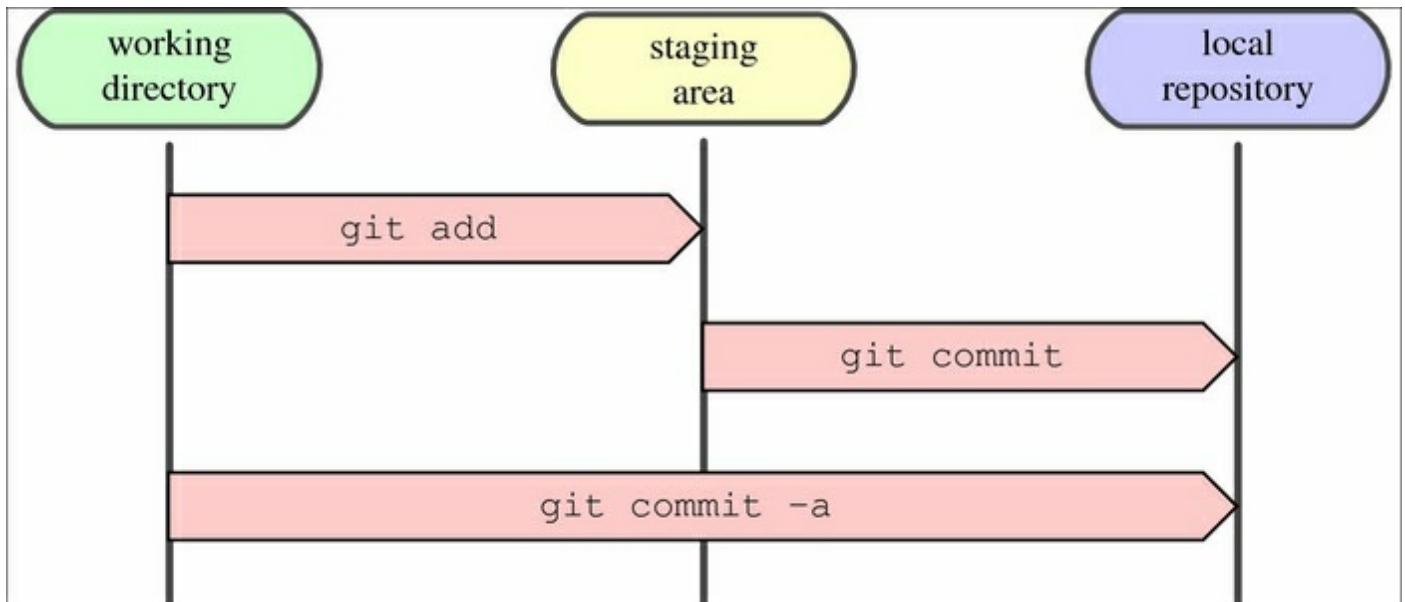


Fig 3. Working directory, staging area, and the local git repository; creating a new commit

The *arrows* show how the Git commands copy contents, for example, `git add` takes the content of the file from the working directory and puts it into the staging area.

Creating a new commit requires the following steps:

1. You make changes to files in your *working directory*, usually modifying them using your favorite editor.
2. You stage the files, adding snapshots of them (their current contents) to your *staging area*, usually with the `git add` command.
3. You create a new revision with the `git commit` command, which takes the files as they are in the staging area and stores that snapshot permanently to your *local repository*.

At the beginning (and just after the commit), the tracked files in the working directory, in the staging area, and in the last commit (the committed version) are identical.

Usually, however, one would use a special shortcut, the `git commit -a` command (which is `git commit --all`), which will take *all the changed*

tracked files, add them to the staging area (as if with `git add -u`, at least in modern Git), and create a new commit (see *Fig 3* of this section). Note that the new files still need to be explicitly `git add` to be tracked, and to be included in the new commit.

Examining the changes to be committed

Before committing the changes and creating a new revision (a new commit), you would want to see what you have done.

Git shows information about the pending changes to be committed in the commit message template, which is passed to the editor, unless you specify the commit message on the command line, for example, with `git commit -m "Short description"`. This template is configurable (refer to [Chapter 10, Customizing and Extending Git](#) for more information).

Note

You can always abort creating a commit by exiting editor without any changes or with an empty commit message (comment lines, that is, lines beginning with `#`, do not count).

In most cases, you would want to examine changes for correctness before creating a commit.

The status of the working directory

The main tool you use to examine which files are in which state: which files have changes, whether there are any new files, and so on, is the `git status` command.

The default output is explanatory and quite verbose. If there are no changes, for example, directly after clone, you could see something like this:

```
$ git status
On branch master
nothing to commit, working directory clean
```

If the branch (you are on the `master` branch in this example) is a local branch intended to create changes that are to be published and to appear in the public repository, and is configured to track its upstream branch, `origin/master`, you would also see the information about the tracked branch:

```
Your branch is up-to-date with 'origin/master'.
```

In further examples, we will ignore it and not include this information.

Let's say you add two new files to your project, a `COPYING` file with the copyright and license, and a `NEWS` file, which is currently empty. In order to begin tracking a new `COPYING` file, you use `git add COPYING`. Accidentally, you remove the `README` file from the working directory with `rm README`. You modify `Makefile` and rename `rand.c` to `random.c` with `git mv` (without modifying it).

The default, long format, is designed to be human-readable, verbose, and descriptive:

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   COPYING
    renamed:   src/rand.c -> src/random.c

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:  Makefile
    deleted:   README

Untracked files:
  (use "git add <file>..." to include in what will be
committed)

    NEWS
```

As you can see, Git does not only describe which files have changed, but also explains how to change their status—either include in the commit, or remove from the set of pending changes (more information about commands in use in `git status` output can be found in [Chapter 4, Managing Your Worktree](#)). There are up to three sections present in the output:

- **Changes to be committed:** This is about the staged changes that would be committed with `git commit` (without the `-a` option). It lists files whose snapshot in the staging area is different from the version from the last commit (`HEAD`).
- **Changes not staged for commit:** This lists the files whose working area contents are different from their snapshot in the staging area. Those changes would not be committed with `git commit`, but would be committed with `git commit -a` as changes in the tracked files.
- **Untracked files:** This lists the files, unknown to Git, which are not ignored (refer to [Chapter 4, Managing Your Worktree](#) for how to use `gitignore`s to make files to be ignored). These files would be added with the bulk `add` command, `git add ..`, in top directory. You can skip this section with `--untracked-files=no` (`-uno` for short).

One does not need to make use of the flexibility that the explicit staging area gives; one can simply use `git add` just to add new files, and `git commit -a` to create the commit from changes to all tracked files. In this case, you would create commit from both the *Changes to be committed* and *Changes not staged for commit* sections.

There is also a terse `--short` output format. Its `--porcelain` version is suitable for scripting because it is promised to remain stable, while `--short` is intended for user output and could change. For the same set of changes, this output format would look something like this:

```
$ git status --short
A  COPYING
M Makefile
D README
R  src/rand.c -> src/random.c
?? NEWS
```

In this format, the status of each path is shown using a two-letter status code. The first letter shows the status of the index (the difference between the staging area and the last commit), and the second letter shows the status of the worktree (the difference between the working area and the staging area):

Symbol	Meaning
	Not updated / unchanged
M	Modified (updated)
A	Added
D	Deleted
R	Renamed
C	Copied

Not all the combinations are possible. Status letters A, R, and C are possible only in the first column, for the status of the index.

A special case, ??, is used for the unknown (untracked) files and !! for ignored files (when using `git status --short --ignored`). Note that not all the possible outputs are described here; the case where we have just done a merge that resulted in merge conflicts is not shown in this table, but is left to be described in [Chapter 7, Merging Changes Together](#).

Examining differences from the last revision

If you want to know not only which files were changed (which you get with `git status`), but also what exactly you have changed, use the `git diff` command:

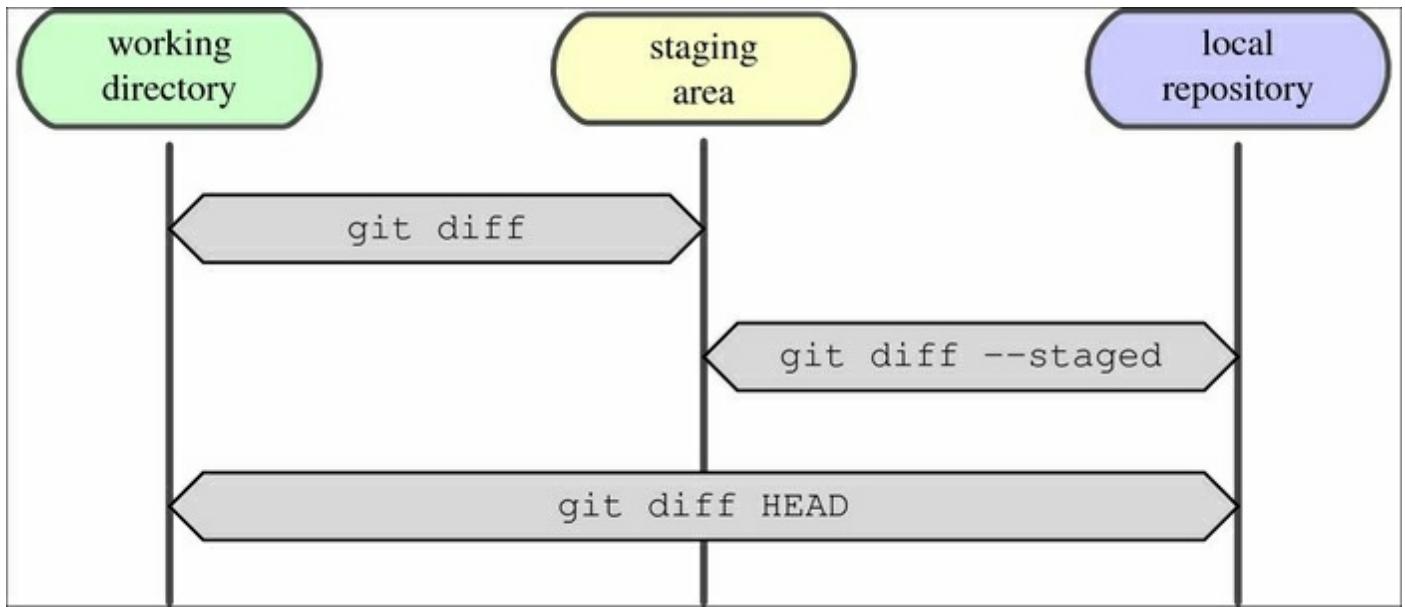


Fig 4. Examining the differences between the working directory, staging area, and local git repository

In the last section, we learned that in Git there are three stages: the working directory, the staging area, and the repository (usually the last commit). Therefore, we have not one set of differences but three, as shown in *Fig 4*. You can ask Git the following questions:

- What have you changed but not yet staged, that is, what are the differences between the staging area and working directory?
- What have you staged that you are about to commit, that is, what are the differences between the last commit (`HEAD`) and staging area?

To see what you've changed but not yet staged, type `git diff` with no other arguments. This command compares what is in your working directory with what is in your staging area. These are the changes that *could* be added, but wouldn't be present if we create commit with `git commit` (without `-a`): *Changes not staged for commit* in the `git status` output.

To see what you've staged that will go into your next commit, use `git diff --staged` (or `git diff --cached`). This command compares what is in your staging area to the content of your last commit. These are the

changes that *would* be added with `git commit` (without `-a`): *Changes to be committed* in the `git status` output. You can compare your staging area to any commit with `git diff --staged <commit>; HEAD` (the last commit) is just the default.

You can use `git diff HEAD` to compare what is in your working directory with the last commit (or arbitrary commit with `git diff <commit>`). These are the changes that would be added with the `git commit -a` shortcut.

If you are using `git commit -a`, and not making use of the staging area, usually it is enough to use `git diff` to check the changes which will be in the next commit. The only issue is the new files that are added with bare `git add`; they won't show in the `git diff` output unless you use `git add --intent-to-add` (or its equivalent `git add -N`) to add new files.

Unified Git diff format

Git, by default and in most cases, will show the changes in **unified diff output format**. Understanding this output is very important, not only when examining changes to be committed, but also when reviewing and examining changes (for example, in code review, or in finding bugs after `git bisect` has found the suspected commit).

Note

You can request only statistics of changes with the `--stat` or `--dirstat` option, or just names of the changed files with `--name-only`, or file names with type of changes with `--name-status`, or tree-level view of changes with `--raw`, or a condensed summary of extended header information with `--summary` (see later for an explanation of what extended header means and what information it contains). You can also request *word diff*, rather than line diff, with `--word-diff`; though this changes only the formatting of chunks of changes, headers and chunk headers remain similar.

Diff generation can also be configured for specific files or types of files

with appropriate **gitattributes**. You can specify external diff helper, that is, the command that describes the changes, or you can specify text conversion filter for binary files (you will learn more about this in [Chapter 4, Managing Your Worktree](#)).

If you prefer to examine the changes in a graphical tool (which usually provides side-by-side diff), you can do it by using `git difftool` in place of `git diff`. This may require some configuration, and will be explained in [Chapter 10, Customizing and Extending Git](#).

Let's take a look at an example of advanced diff from Git project history. Let's use the diff from the commit `1088261f` from the `git.git` repository. You can view these changes in a web browser, for example, on GitHub; this is the third patch in this commit:

```
diff --git a/builtin-http-fetch.c b/http-fetch.c
similarity index 95%
rename from builtin-http-fetch.c
rename to http-fetch.c
index f3e63d7..e8f44ba 100644
--- a/builtin-http-fetch.c
+++ b/http-fetch.c
@@ -1,8 +1,9 @@
 #include "cache.h"
 #include "walker.h"

-int cmd_http_fetch(int argc, const char **argv, const char
 *prefix)
+int main(int argc, const char **argv)
{
+    const char *prefix;
    struct walker *walker;
    int commits_on_stdin = 0;
    int commits;
@@ -18,6 +19,8 @@ int cmd_http_fetch(int argc, const char
 **argv,
            int get_verbosely = 0;
            int get_recover = 0;

+    prefix = setup_git_directory();
+
        git_config(git_default_config, NULL);
```

```
while (arg < argc && argv[arg][0] == '-') {
```

Let's analyze this patch line after line:

- The first line, `diff --git a/builtin-http-fetch.c b/http-fetch.c`, is a *git diff* header in the form `diff --git a/file1 b/file2`. The `a/` and `b/` filenames are the same unless rename or copy is involved (such as in our case), even if the file is added or deleted. The `--git` option means that `diff` is in the `git diff` output format.
- The next lines are one or more extended header lines. The first three lines in this example tell us that the file was renamed from `builtin-http-fetch.c` to `http-fetch.c` and that these two files are 95% identical (which information was used to detect this rename):

```
similarity index 95%
rename from builtin-http-fetch.c
Rename to http-fetch.c
```

Note

Extended header lines describe information that cannot be represented in an ordinary unified diff (except for information that file was renamed). Besides similarity (or dissimilarity) score like in example they can describe the changes in file type (example from non-executable to executable).

- The last line in extended diff header, which, in this example is `index f3e63d7..e8f44ba 100644` tells us about the mode of given file (`100644` means that it is an ordinary file and not a symbolic link, and that it doesn't have executable permission bit; these three are only file permissions tracked by Git), and about shortened hash of pre-image (the version of the file before the given change) and post-image (the version of the file after the change). This line is used by `git am --3way` to try to do a three-way merge if the patch cannot be applied itself. For the new files, pre-image hash is `0000000`, the same for the deleted files with post-image hash.
- Next is the unified diff header, which consists of two lines:

```
--- a/builtin-http-fetch.c
```

```
+++ b/http-fetch.c
```

- Compared to the `diff -U` result, it doesn't have from-file-modification-time or to-file-modification-time after source (pre-image) and destination or target (post-image) filenames. If the file was created, the source would be `/dev/null`; if the file was deleted, the target would be `/dev/null`.

Note

If you set the `diff.mnemonicPrefix` configuration variable to `true`, in place of the `a/` prefix for pre-image and `b/` for post-image in this two-line header, you can instead have the `c/` prefix for commit, `i/` for index, `w/` for worktree, and `o/` for object, respectively, to show what you compare.

- Next comes one or more hunk of differences; each hunk shows one area where the files differ. Unified format hunks start with the line describing where the changes were in the file:

```
@@ -1,8 +1,9 @@
```

This line is in the format `@@ from-file-range to-file-range @@`. The from-file-range is in the form `-<start line>, <number of lines>`, and to-file-range is `+<start line>, <number of lines>`. Both start-line and number-of-lines refer to the position and length of hunk in pre-image and post-image, respectively. If number-of-lines is not shown, it means that it is 0. In this example, the changes, both in pre-image (file before the changes) and post-image (file after the changes) begin at the first line of the file, and the fragment of code corresponding to this hunk of diff has 8 lines in pre-image, and 9 lines in post-image (one line is added). By default, Git will also show three unchanged lines surrounding changes (three context lines). Git will also show the function where each change occurs (or equivalent, if any, for other types of files; this can be configured with `.gitattributes`); it is like the `-p` option in GNU diff:

```
@@ -18,6 +19,8 @@ int cmd_http_fetch(int argc, const char
```

- Next is the description of where and how files differ. The lines

common to both the files begin with a space (" ") indicator character. The lines that actually differ between the two files have one of the following indicator characters in the left print column:

- +: A line was added here to the second file
- -: A line was removed here from the first file

Note

Note that the changed line is denoted as removing the old version and adding the new version of the line.

In the plain word-diff format, instead of comparing file contents line by line, added words are surrounded by {+ and +}, while removed by [- and -].

- If the last hunk includes, among its lines, the very last line of either version of the file, and that last line is incomplete, (which means that the file does not end with the end-of-line character at the end of hunk) you would find:

\ No newline at end of file

This situation is not present in the presented example.

So, for the example used here, first chunk means that `cmd_http_fetch` was replaced by `main` and the `const char *prefix;` line was added:

```
#include "cache.h"
#include "walker.h"

-int cmd_http_fetch(int argc, const char **argv, const char
*prefix)
+int main(int argc, const char **argv)
{
+    const char *prefix;
        struct walker *walker;
        int commits_on_stdin = 0;
        int commits;
```

See how for the replaced line, the old version of the line appears as removed (-) and the new version as added (+).

In other words, before the change, the appropriate fragment of the file, that was then named `builtin-http-fetch.c`, looked similar to the following:

```
#include "cache.h"
#include "walker.h"

int cmd_http_fetch(int argc, const char **argv, const char
*prefix)
{
    struct walker *walker;
    int commits_on_stdin = 0;
    int commits;
```

After the change, this fragment of the file that is now named `http-fetch.c`, looks similar to this instead:

```
#include "cache.h"
#include "walker.h"

int main(int argc, const char **argv)
{
    const char *prefix;
    struct walker *walker;
    int commits_on_stdin = 0;
    int commits;
```

Selective commit

Sometimes, after examining the pending changes as explained, you realize that you have two (or more) unrelated changes in your working directory that should belong to two different logical changes; it is the tangled working copy problem. You need to put those unrelated changes into separate commits, as separate changesets. This is the type of situation that can occur even when trying to follow the best practices.

One solution is to create commit as-is, and fix it later (split it in two). You can read how to do this in [Chapter 8, Keeping History Clean](#).

Sometimes, however, some of the changes are needed now, and shipped immediately (for example bug fix to a live website), while the rest of the

changes are work in progress, not ready. You need to tease those changes apart into two separate commits.

Selecting files to commit

The simplest situation is when these unrelated changes touch different files. For example, if the bug was in the `view/entry tmpl` file and only in this file, and there were no other changes to this file, you can create a bug fix commit with the following command:

```
$ git commit view/entry tmpl
```

This command will ignore changes staged in the index (what was in the staging area), and instead record the current contents of a given file or files (what is in the working directory).

Interactively selecting changes

Sometimes, however, the changes cannot be separated in this way. The changes to the file are tangled together. You can try to tease them apart by giving the `--interactive` option to `git commit`:

```
$ git commit --interactive
      staged      unstaged path
1: unchanged      +3/-2 Makefile
2: unchanged      +64/-1 src/rand.c

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

Here, Git shows us the status and the summary of changes to the working area (`unstaged`) and to the staging area / the index (`staged`)—the output of the `status` subcommand. The changes are described by the number of added and deleted files (similar to what the `git diff --numstat` command shows):

```
What now> h
status           - show paths with changes
update          - add working tree state to the staged set of
```

```

changes
revert      - revert staged set of changes back to the HEAD
version
patch       - pick hunks and update selectively
diff        - view diff between HEAD and index
add untracked - add contents of untracked files to the staged
set of changes
*** Commands ***
1: status      2: update      3: revert      4: add
untracked
5: patch      6: diff        7: quit        8: help

```

To tease apart changes, you need to choose the `patch` subcommand (for example, with 5 or s). Git will then ask for the files with the `Update>>` prompt; you then need to select the files to selectively update with their numeric identifiers, as shown in the status, and type `return`. You can say * to select all the files possible. After making the selection, end it by answering with an empty line. (You can skip directly to patching files with the `--patch` option.)

Git will then display all the changes to the specified files on a hunk-by-hunk basis, and let you choose, among others, one of the following options for each hunk:

```

y - stage this hunk
n - do not stage this hunk
q - quit; do not stage this hunk or any of the remaining
ones

s - split the current hunk into smaller hunks
e - manually edit the current hunk
? - print help

```

The hunk output and the prompt look similar to this:

```

@@ -16,7 +15,6 @@
int main(int argc, char *argv[])
{
    int max = atoi(argv[1]);
+
    srand(time(NULL));
    int result = random_int(max);
    printf("%d\n", result);
}

```

```
Stage this hunk [y,n,q,a,d,/,,j,J,g,e,?] ? y
```

In many cases, it is enough to simply select which of those hunks of changes you want to have in the commit. In extreme cases, you can split a chunk into smaller pieces, or even manually edit the diff.

Creating a commit step by step

Interactively selecting changes to commit with `git commit --interactive` doesn't unfortunately allow to test the changes to be committed. You can always check that everything works after creating a commit (compile and/or run tests), and then amend it if there are any errors. There is, however, an alternative solution.

You can prepare commit by putting the pending changes into the staging area with `git add --interactive`, or an equivalent solution (like graphical Git commit tool for Git, for example, `git gui`). The interactive commit is just a shortcut for interactive add followed by commit, anyway. Then you should examine these changes with `git diff --cached`, modifying them as appropriate with `git add <file>`, `git checkout <file>`, and `git reset <file>`.

In theory, you should also test these changes whether they are correct, checking that at least they do not break the build. To do this, first use `git stash save --keep-index` to save the current state and bring the working directory to the state prepared in the staging area (the index). After this command, you can run tests (or at least check whether the program compiles and doesn't crash). If tests pass, you can then run `git commit` to create a new revision. If tests fail, you should restore the working directory while keeping the staging area state with the `git stash pop --index` command; it might be required to precede it with `git reset --hard`. The latter might be needed because Git is overly conservative when preserving your work, and does not know that you have just stashed. First, there are uncommitted changes in the index prevent Git from applying the stash, and second, the changes to the working directory are the same as stashed, so of course they would conflict.

You can find more information about stashes, including how they work, in [Chapter 4, Managing Your Worktree](#).

Amending a commit

One of the better things in Git is that you can undo almost anything; you only need to know how. No matter how carefully you craft your commits, sooner or later, you'll forget to add a change, or mistype the commit message. That's when the `--amend` flag of the `git commit` command comes in handy; it allows you to change the very last commit really easily. Note that you can also amend the merge commits (for example, fix a merging error).

Note

If you want to change a commit deeper in history (assuming that it was not published, or at least, there isn't anyone who based their work on the old version of the said commit), you need to use **interactive rebase** or some specialized tool, such as **StGit** (*a patch stack management interface* on top of Git). Refer to [Chapter 8, Keeping History Clean](#), for more information.

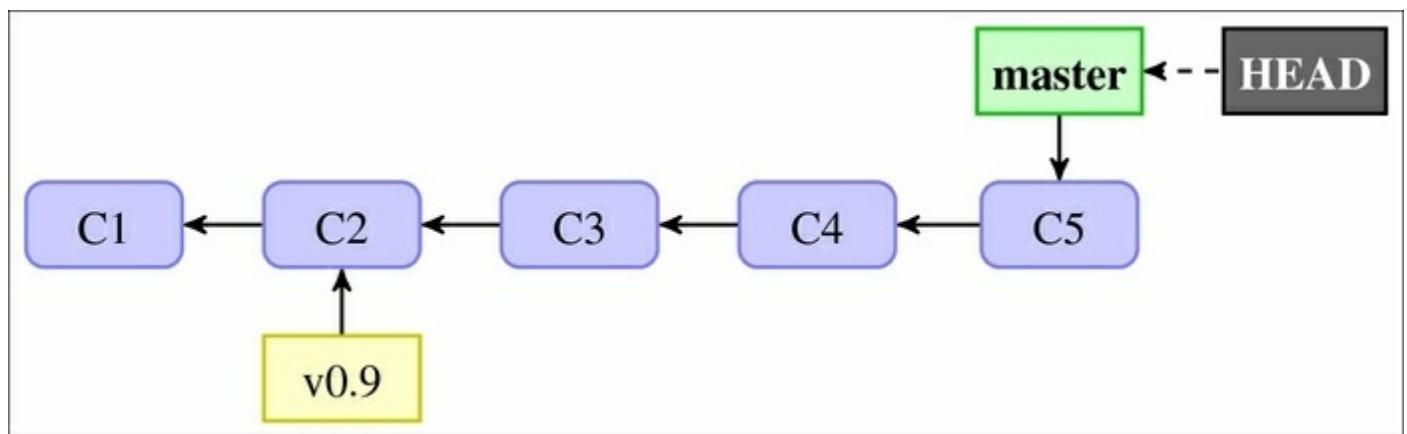


Fig 5. The DAG of revisions, C1 to C2, before amending a topmost (most recent) and currently checked out commit, which is named C5. Here, we have used numbers instead of SHA-1 to be able to indicate related commits.

If you just want to correct the commit message, you simply *commit again*, without any staged changes, and fix it (note that we use `git commit` without the `-a` / `--all` flag):

```
$ git commit --amend
```

If you want to add some more changes to that last commit, you can simply stage them as normal with `git add` and then commit again as shown in the preceding example, or make the changes and use `git commit -a --amend`:

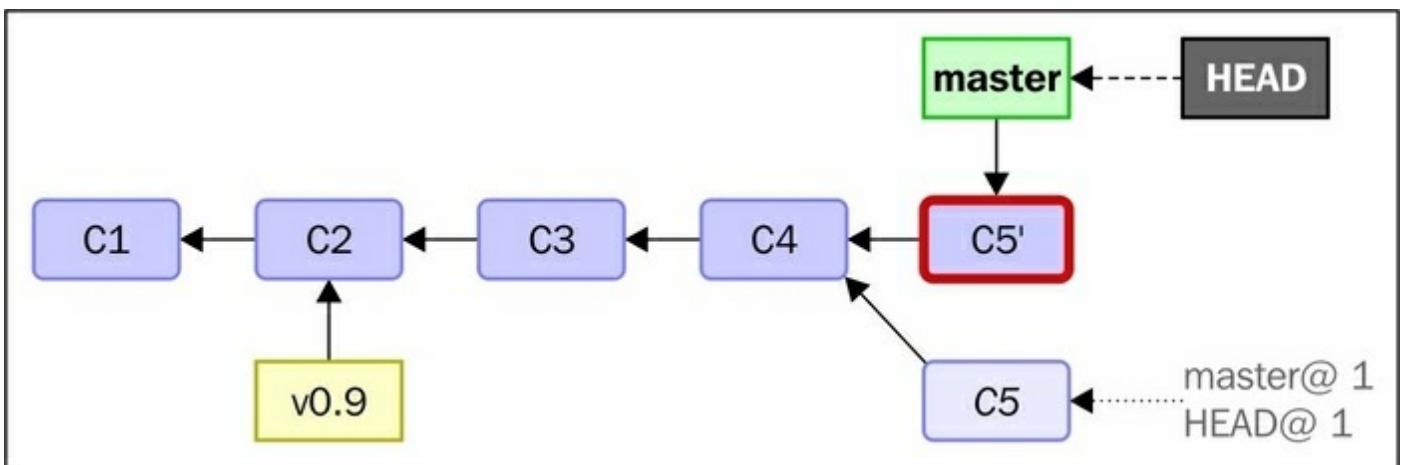


Fig 6. The DAG of revisions after amending the last commit (revision C5) on Fig 5. Here, the new commit C5 is old commit C5 with changes (amended); it replaces old commit place in history.

There is a very important caveat: you should *never* amend a commit that has already been published! This is because amend effectively produces a completely new commit object that replaces the old one, as can be seen on *Fig 6*. If you're the only person who had this commit, doing this is safe. However, after publishing the original commit to a remote repository, other people might already have based their new work on that version of the commit. Replacing the original with an amended version will cause problems downstream. You will find more about this issue in [Chapter 8, Keeping History Clean](#).

If you try to push (publish) a branch with the published commit

amended, Git would prevent overwriting the published history, and ask to force push if you really want to replace the old version (unless you configure it to force push by default). The old version of commit before amending would be available in the branch reflog and in the HEAD reflog; for example, just after *amend*, it would be available as `@{1}`. Git would keep the old version for a month, by default, unless manually purged.

Working with branches

Branches are symbolic names for lines of development. In Git, each branch is realized as a named *pointer* (reference) to some commit in the DAG of revisions, so it is called **branch head**.

Tip

The representation of branches in Git

Git uses currently two different on-disk representations of branches: the *loose* format and the *packed* format.

Take, for example, the `master` branch (which is the default name of a branch in Git; you start on this branch in the newly-created repository). In *loose* format (which takes precedence), it is represented as the one-line file, `.git/refs/heads/master` with textual hexadecimal representation of SHA-1 tip of the branch. In the *packed* format, it is represented as a line in the `.git/packed-refs` file, connecting SHA-1 identifier of top commit with the fully qualified branch name.

The (named) line of development is the set of all revisions that are reachable from the branch head. It is not necessarily a straight line of revisions, it can fork and join.

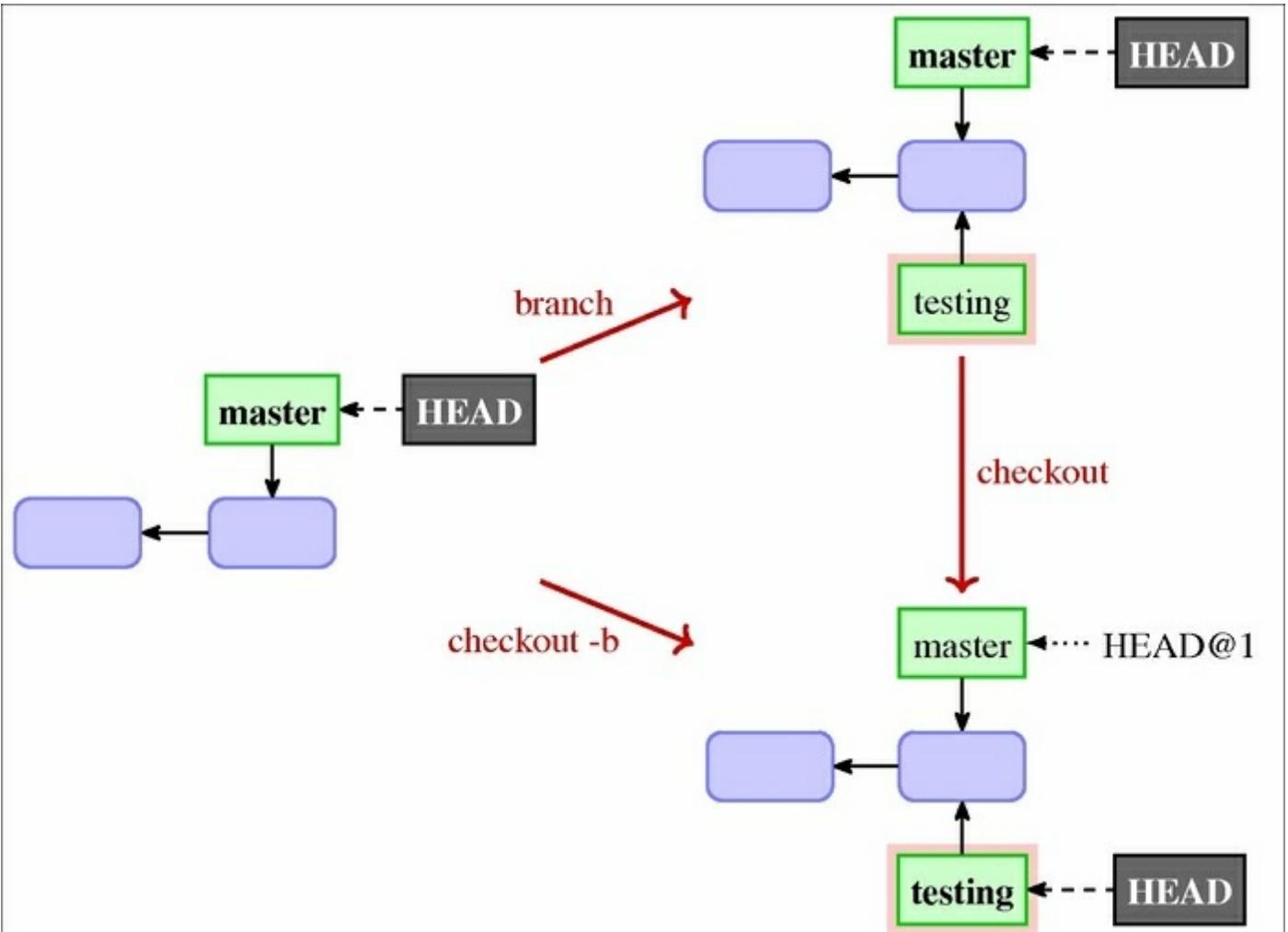


Fig 7. Creating a new testing branch and switching to it, or creating a new branch and switching to it at once (with one command)

Creating a new branch

You can create a new branch with the `git branch` command; for example, to create a new branch `testing` starting from the current branch (see the top right part of *Fig 7*), run:

```
$ git branch testing
```

What happens here? Well, this command creates a new pointer (a new reference) for you to move around. You can give an optional parameter to this command if you want to create the new branch pointing to some other commit.

Note, however, that the `git branch` command would not change the position of the `HEAD` (the symbolic reference pointing to current branch), and would not change the contents of the working directory.

If you want to create a new branch and switch to it (to start working on new branch immediately), you can use the following shortcut:

```
$ git checkout -b testing
```

If we create a new branch at the current state of repository, the `checkout -b` command differs only in that it also moves the `HEAD` pointer; see transition from left-hand side to the bottom-right in *Fig 7*.

Creating orphan branches

Sometimes you might want to create a new unconnected *orphan* branch in your repository. Perhaps you want to store the generated documentation for each release to make it easy for users to get readable documentation (for example, as man pages or HTML help) without requiring to install conversion tools or renderers (for example, AsciiDoc parser). Or, you might want to store web pages for a project in the same repository as project; that is what GitHub project pages use. Perhaps you want to open source your code, but you need to clean up the code first (for example, because of copyrights and licensing).

One solution is to create a separate repository for the contents of an orphan branch, and fetch from it into some remote-tracking branch. You can then create a local branch based on it.

You can also do this with:

```
$ git checkout --orphan gh-pages
Switched to a new branch 'gh-pages'
```

This reproduces somewhat the state just after `git init`: the `HEAD` symref points to the `gh-pages` branch, which does not exist yet; it will be created on the first commit.

If you want to start with clean state, like with GitHub Pages, you would

also need to remove the contents of the start point of branch (which defaults to `HEAD`, that is, to current branch and to the current state of the working directory), for example with:

```
$ git rm -rf .
```

In the case of open sourcing code with proprietary parts to be excluded (*orphan* branch is not to bring this proprietary code accidentally to the open source version on merging), you would want to carefully edit the working directory instead.

Selecting and switching to a branch

To switch to an existing local branch, you need to run the `git checkout` command. For example, after creating the `testing` branch, you can switch to it with the following command:

```
$ git checkout testing
```

It is shown in *Fig 7* as the vertical transition from the top-right to bottom-right state.

Obstacles to switching to a branch

When switching to a branch, Git also checks out its contents into the working directory. What happens then if you have uncommitted changes (which are not considered by Git to be on any branch)?

Note

It is a good practice to switch branch in a clean state, stashing away changes or creating a commit, if necessary. Checking out a branch with uncommitted changes is useful only in a few rare cases, some of which are described in the following section.

If the difference between the current branch and the branch you want to switch to does not touch the changed files, the uncommitted changes are moved to the new branch. This is very useful if you started working on something, and only later realized that it would be better to do this work

in a separate feature branch.

If uncommitted changes conflict with changes on the given branch, Git will refuse to switch to the said branch, to prevent you from losing your work:

```
$ git checkout other-branch  
error: Your local changes to the following files would be  
overwritten by checkout:  
      file-with-local-changes  
Please, commit your changes or stash them before you can switch  
branches.
```

In such situation you have a few possible different solutions:

- You can *stash away* your changes, and restore them when you come back to the branch you were on (this is usually the preferred solution). Or you can simply create a temporary commit of the work in progress with those changes, and then either amend the commit or rewind the branch when you get back to it.
- You can try to move your changes to the new branch by *merging*, either with `git branch --merge` (which would do the three-way merge between the current branch, the contents of your working directory with unsaved changes, and the new branch), or by stashing away your changes before checkout and then unstashing them after a switch.
- You can also *throw away* your changes with `git checkout --force`.

Anonymous branches

What happens if you try to check out something that is not a local branch: for example an arbitrary revision (like `HEAD^`), or a tag (like `v0.9`), or a remote-tracking branch (for example, `origin/master`)? Git assumes that you need to be able create commits on top of the current state of the working directory.

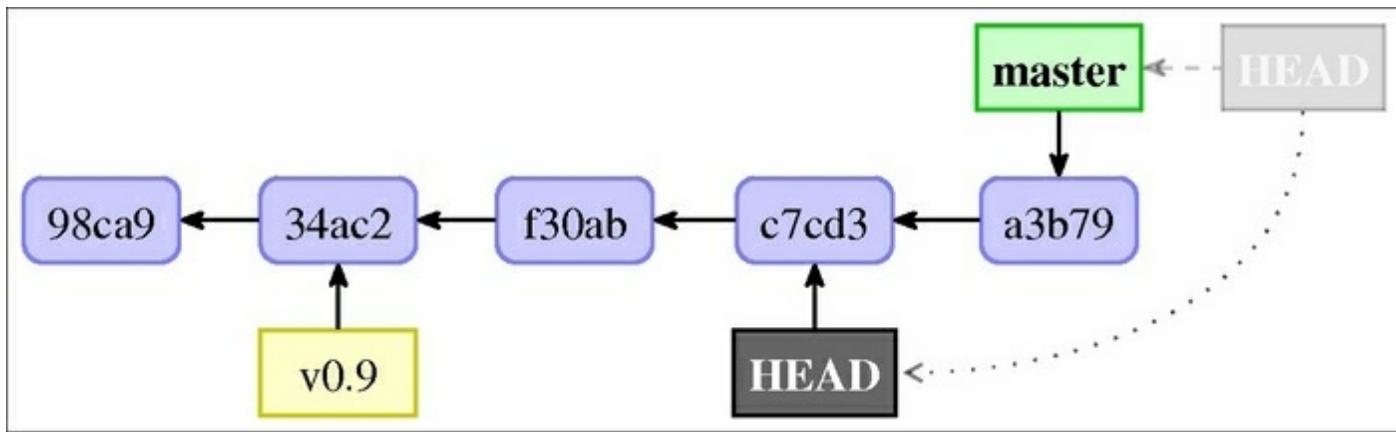


Fig 8. The result of checking out non-branch, the state after Git checkout HEAD command, detached HEAD, or anonymous branch

Older Git refused to switch to non-branch. Nowadays, Git will create an anonymous branch by detaching HEAD pointer and making it refer directly to a commit, rather than being a symbolic reference to a branch, see *Fig 8* for an example. To create an anonymous branch at the current position explicitly, you can use the `--detach` option to the checkout command. The detached HEAD state is shown in branch listing as (*no branch*) in older versions of Git, or (*detached from HEAD*) or (*HEAD detached at ...*) in newer.

If you did detach HEAD by mistake, you can always go back to the previous branch with (here "`-`" means the name of previous branch):

```
$ git checkout -
```

As Git informs you when creating a detached branch, you can always give a name to the anonymous branch with `git checkout -b <new-name>`.

Git checkout DWIM-mery

There is a special case of checking out something that is not a branch. If you check out remote-tracking branch (for example, `origin/next`) by its short name (in this case, `next`), as if it was a local branch, Git would assume that you meant to create new contents on top of the remote-

tracking branch state, and will do what it thinks you need. **Do What I Mean (DWIM)** will create a new local branch, tracking the remote-tracking branch.

This means that:

```
$ git checkout next
```

Is equivalent to:

```
$ git checkout -b next --track origin/next
```

Git will do it only if there are no ambiguities: the local branch must not exist (otherwise the command would simply switch to local branch given), and there can be only one remote-tracking branch that matches. This can be checked by running `git show-ref next` (using the short name) and verifying that it returns only one line, with remote-tracking branch info (the last can be recognized by the `refs/remotes/` prefix in ref name).

Listing branches

If you use the `git branch` commands without any other arguments, it would list all the branches, marking the current branch with asterisk, that is, `*`.

Note

This command is intended for the end user; its output may change in the future version of Git. To find out programmatically, in a shell script:

- To get the name of the current branch, use `git symbolic-ref HEAD`.
- To find SHA-1 of the current commit, use `git rev-parse HEAD`.
- To list all the branches, use `git show-ref` or `git for-each-ref`.

They are all **plumbing**, that is, commands intended for use in scripts.

You can request more information with `-v` (`--verbose`) or `-vv`. You can also limit branches shown to only those matching given shell wildcard

with `git branch --list <pattern>` (quoting pattern to prevent its expansion by shell, if necessary).

Querying information about remotes, which includes the list of remote branches, by using `git remote show`, is described in [Chapter 6, Advanced Branching Techniques](#).

Rewinding or resetting a branch

What to do if you want to abandon the last commit, and rewind (reset) the current branch to its previous position? For this, you need to use the `reset` command. It would change where the current branch points to. Note that unlike the `checkout` command, the `reset` command does not change the working directory by default; you need to use instead `git reset --keep` (to try to keep the uncommitted changes) or `git reset --hard` (to drop them).

The `reset` command, and its effects on the working area, will be explained in more detail in [Chapter 4, Managing Your Worktree](#).

Fig 9 shows the differences between the `checkout` and `reset` commands, when given the branch and non-branch argument. In short, `reset` always changes where the current branch points to (moves the ref), while `checkout` either switches branch, or detaches `HEAD` at a given revision if it is given non-branch:

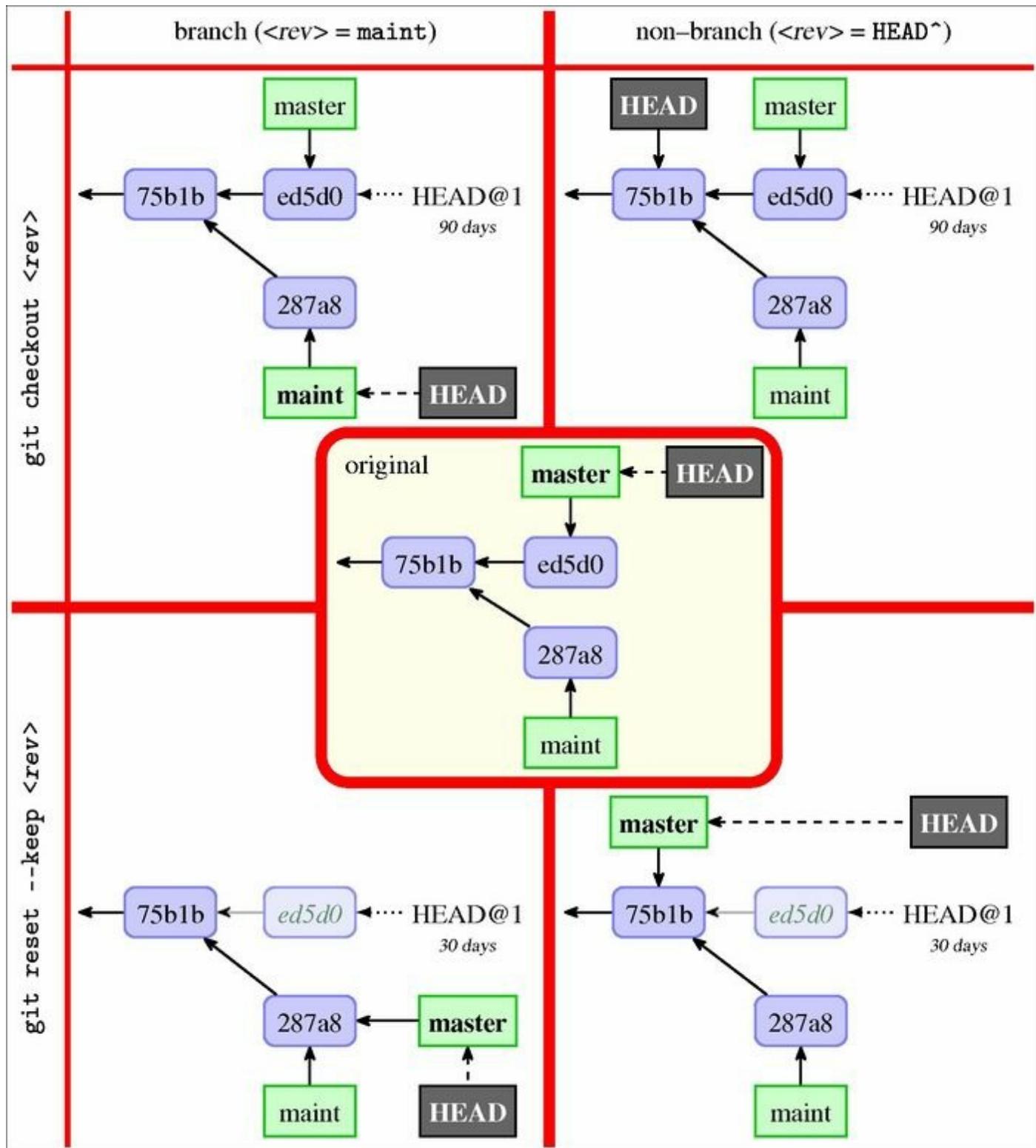


Fig 9. A table comparing the checkout and reset commands with either branch (for example, `maint`) and non-branch revision (for example, `HEAD^`) as arguments.

In the preceding figure, for example, the left-top graph of the revision

shows the result of running of the `git checkout maint` command, starting from the state given by the graphs in the centre.

Deleting a branch

As in Git, a branch is just a pointer, and an external reference to the node in the DAG of revisions, deleting a branch is just deleting a pointer:

Note

Actually deleting a branch also removes, irretrievably, (at least, in the current Git version) the reflog for the branch being deleted, that is, the log of its local history.

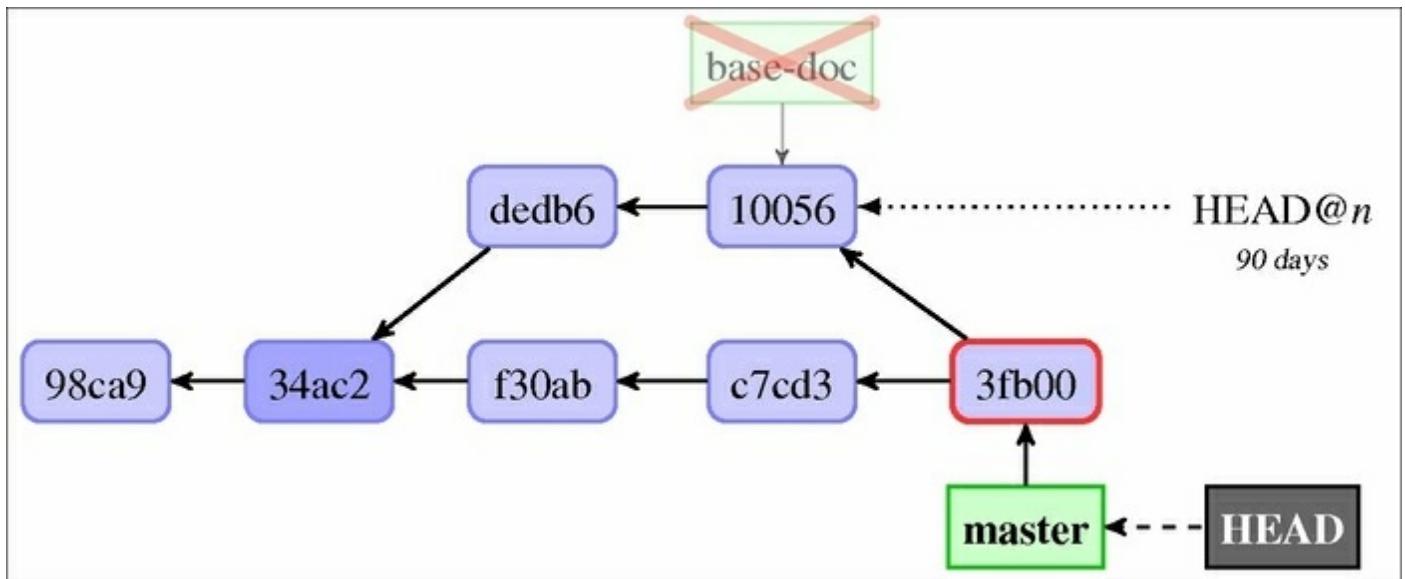


Fig 10. Deleting just merged in base-doc branch with git branch -d base-doc, when we are on a branch (master here) that includes it

You can do this with `git branch -d`. There is, however, one issue to consider—what happens if you delete a branch, and there is no other reference to the part of project history it pointed to? Those revisions will become unreachable and Git would delete them after the `HEAD` reflog expires (which, with default configuration, is after 30 days).

That is why Git would allow you to delete only the completely merged-

in branch, whose all commits are reachable from `HEAD` as in *Fig 10* (or is reachable from its upstream branch, if it exists).

To delete a branch that was not merged in, risking parts of the DAG becoming unreachable, you need a stronger command, namely, `git branch -D` (Git will suggest this operation when refusing to delete a branch); see *Fig 11*:

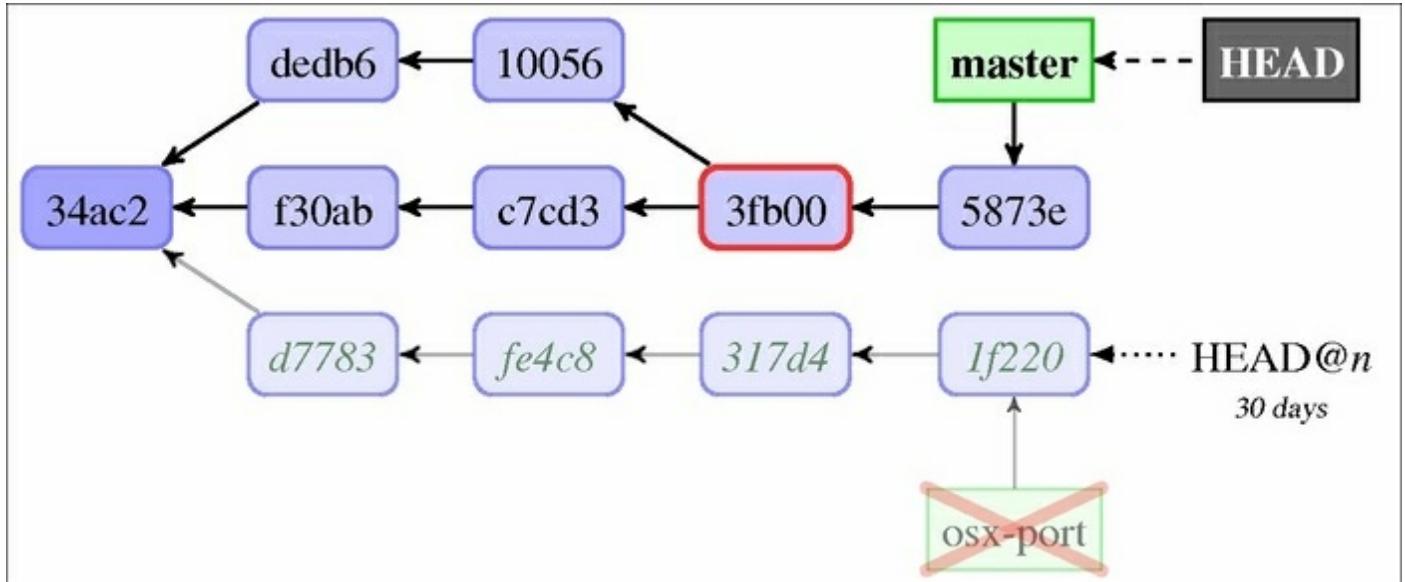


Fig 11. Deleting the unmerged osx-port branch with git branch -D osx-port

You can check if the branch was merged in into any other branch, by checking whether `git branch --contains <branch>` shows anything. You cannot delete the current branch.

Changing the branch name

Sometimes the name chosen for a branch needs to be changed. This can happen, for example, if the scope of the feature branch changed during the development.

You can rename a branch with `git branch -m` (use `-M` if target name exists and you want to overwrite it); it will rename a branch and move the corresponding reflog (and add rename operation to the reflog), and

change the branch in all of its configuration (its description, its upstream, and so on).

Summary

In this chapter, we learnt how to develop with Git and add to the project history by creating new commits and new lines of development (branches). We know what it means to create a commit, to amend a commit, to create a branch, to switch a branch, to rewind a branch, and to delete a branch from the point of view of the Directed Acyclic Graph of revisions.

This chapter shown a very important Git feature—the staging area for creating commits, also known as the index. This is what makes it possible to untangle the changes to the working directory by selectively and interactively choosing what to commit.

We learnt how to examine the changes to the working area before creating a commit. This chapter described, in detail, the extended unified diff format that Git uses to describe the changes.

We also learnt about the concept of detached HEAD (or anonymous branch) and of orphan branches.

In the next chapter, *Managing Your Worktree*, we will learn how to use Git to prepare new commits and how to configure it to make our work easier. We will also learn how to examine, search, and study the contents of the working directory, the staging area, and the project history. We will also see how to use Git to deal with interruptions and recover from mistakes.

Chapter 4. Managing Your Worktree

The previous chapter, *Developing with Git*, described how to use Git for development, including how to create new revisions. Now we will focus on learning how to manage your working directory (worktree) to prepare contents for a new commit. This chapter will teach you how to manage your files, in detail. It will show how to care for files that require special handling, introducing the concepts of ignored files and file attributes.

You will also learn how to fix mistakes in handling files, both in the working directory and in the staging area; and how to fix the latest commit. You will find out how to safely handle interruptions in the workflow with stash and multiple working directories.

The previous chapter taught you how to examine changes. Here you will learn how to undo and redo those changes selectively, and how to view different versions of a file.

This chapter will cover the following topics:

- Ignoring files: marking files as intentionally not under version control
- File attributes: path-specific configuration
- Handling text and binary files
- End of line conversion of text files, for repository portability
- Using various modes of the `git reset` command
- Stashing away your changes to handle interruptions
- Searching and examining files in any place
- Resetting files and reverting file changes interactively
- Cleaning the working area by removing untracked files

Ignoring files

Your files inside your working area (also known as the **worktree**) can be

either *tracked* or *untracked* by Git. Tracked files, as the name suggests, are whose changes Git will follow. For Git, if a file is present in the *staging area* (also known as **the index**), it will be tracked and—unless specified otherwise—it will be a part of the next revision. You **add** files to be tracked so as to have them as a part of the project history.

Note

The index, or the staging area, is used not only for Git to know which files to track, but also as a kind of a scratchpad to create new commits, as described in [Chapter 3, *Developing with Git*](#), and to help resolve merge conflicts, as shown in [Chapter 7, *Merging Changes Together*](#).

Often you will have some individual files or a class of files that you never want to be a part of the project history, and never want to track. These can be your editor backup files, or automatically generated files produced by the project's build system.

You don't want Git to automatically add such files, for example, when doing *bulk add* with `git add :/` (adding the entire working tree), `git add .` (adding the current directory), or when updating the index to the worktree state with `git add --all`. Quite the opposite: you want Git to actively prevent from accidentally adding them. You also want such files to be absent from the `git status` output, as there can be quite a number of them. They could drown out legitimate new *unknown* files there otherwise. You want such files to be intentionally untracked: *ignored*.

Tip

Un-tracking and re-tracking files

If you want to start ignoring a file that was formerly tracked, for example when moving from hand-generated HTML file to using a lightweight markup language such as Markdown instead, you usually need to **un-track** the file without removing it from the working directory, while adding it to the list of ignored files. You can do this with `git rm --cached <file>`.

To add (start tracking) an intentionally untracked (that is, ignored) file, you need to use `git add -f`.

Marking files as intentionally untracked

In such case, you can add a shell glob pattern to match files that you want to have ignored by Git to one of the **gitignore** files, one pattern per line:

- The per-user file that can be specified by the configuration variable `core.excludesFile`, which by default is `$XDG_CONFIG_HOME/git/ignore`. This in turn defaults to the `$HOME/.config/git/ignore` if `$XDG_CONFIG_HOME` environment variable is not set or empty.
- The per-local repository `$GIT_DIR/info/exclude` file in the administrative area of the local clone of the repository.
- The `.gitignore` files in the working directories of a project; these are usually tracked and thus shared among developers.

Some commands, such as `git clean`, also allow us to specify ignore patterns from a command line.

When deciding whether to ignore a path, Git checks all those sources in the order specified on preceding list, with the last matching pattern deciding the outcome. The `.gitignore` files are checked in order, starting from the top directory of the project down to the directory of files to be examined.

To make `gitignore` files more readable you can use blank lines to separate groups of files (a blank line matches no files). You can also describe patterns or groups of patterns with comments; a line starting with `#` serves as one (to ignore a pattern beginning with the hash character, `#`, escape the first hash character with a backslash `\`, for example, `\#*#`). Trailing spaces (at the end of the line) are ignored unless escaped with a backslash `\`.

Each line in the `gitignore` file specifies a Unix glob pattern, a shell wildcard. The `*` wildcard matches zero or more characters (any string),

while the ? wildcard matches any single character. You can also use character classes with brackets [. . .]. Take for example the following list of patterns:

```
*.[oa]  
*~
```

Here the first line tells Git to ignore all files with the .a or .o extension—archive (for example, a static library) and object files that may be the products of compiling your code. The second line tells Git to ignore all files ending with a tilde, ~; this is used by many Unix text editors to mark temporary backup files.

If the pattern does not contain a slash /, which is a directory (path component) separator, Git treats it as a shell glob and checks file name or directory name for a match, starting at appropriate depth, for example the .gitignore file location, or the top level of the repository. The exception is patterns ending with slash /—which is used to have the pattern matched against directories only—but otherwise the trailing slash is removed. A leading slash matches the beginning of the path name. This means the following:

- Patterns not containing a slash match everywhere in the repository; one can say that the pattern is recursive.

For example, the * .o pattern matches object files anywhere, both in the gitignore file level and in subdirectories: file.o, obj/file.o, and so on.

- Patterns ending with a slash match only directories, but are otherwise recursive (unless they contain other slashes).

For example, the auto/ pattern will match the top-level auto directory and for example src/auto, but will not match the auto file (or a symbolic link either).

- To *anchor* a pattern and make it non-recursive, add a leading slash.

For example the /TODO file will ignore the current-level TODO file, but not files in subdirectories, for example src/TODO.

- Patterns containing a slash are anchored and non-recursive, and wildcard characters (*, ?, a character class such as [ao]) do not match the directory separator that is slash. If you want to match any number of directories, use two consecutive asterisks ** in place of the path component (which means **/foo, foo/**, and foo/**/bar).

For example, `doc/*.html` matches `doc/index.html` file but not `doc/api/index.html`; to match HTML files anywhere inside the `doc` directory you can use the `doc/**/*.*.html` pattern (or put the `*.*.html` pattern in the `doc/.gitignore` file).

You can also negate a pattern by prefixing it with an exclamation mark !; any matching file excluded by the earlier rule is then included (non-ignored) again. For example to ignore all generated HTML files, but include one generated by hand, you can put the following in the `gitignore` file:

```
# ignore html files, generated from Asciidoc sources
*.html
# except for the files below which are generated by hand
!welcome.html
```

Note however that for performance reasons Git doesn't go into excluded directories, and (up till Git 2.7) this meant that you cannot re-include a file if a parent directory is excluded. This means that to ignore everything except for the subdirectory, you need to write the following:

```
# exclude everything except directory t0001/bin
/*
!/t0001
/t0001/*
!/t0001/bin
```

To match a pattern beginning with !, escape it with a backslash—for example, `\!important!.md` to match `!important!.md`.

Which types of file should be ignored?

Now that we know *how* to mark files as intentionally untracked

(ignored), there is the question of *which* files (or classes of files) should be marked as such. Another issue is *where*, in which of the three `gitignore` files, should we add a pattern for ignoring specific types of file?

First, you should never track automatically generated files (usually generated by the build system of a project). If you add them to the repository, there is a high chance that they will get out of sync with their source. Besides, they are not necessary, as you can always re-generate them. The only possible exception is generated files where the source changes rarely, and generating them requires extra tools that developers might not have (if the source changes more often, you can use an orphan branch to store these generated files, and refresh this branch only at release time).

Those are the files that all developers will want to ignore. Therefore they should go into a tracked `.gitignore` file. The list of patterns will be version-controlled and distributed to other developers via a clone. You can find a collection of useful `.gitignore` templates for different programming languages at <https://github.com/github/gitignore>.

Second, there are temporary files and by-products specific to one user's toolchain; those should usually not be shared with other developers. If the pattern is specific to both the repository and the user, for example, auxiliary files that live inside the repository but are specific to the workflow of a user (for example, to the IDE used for the project), it should go into the per-clone `$GIT_DIR/info/exclude` file.

Patterns which the user wants to ignore in all situations, not specific to the repository (or to the project), should generally go into a file specified by the `core.excludesFile` config variable, set in the per-user (global) config file `~/.gitconfig` (or `~/.config/git/config`). This is usually by default `~/.config/git/ignore`.

Note

The per-user ignore file cannot be `~/.gitignore`, as this would be the in-

repository `.gitignore` file for the versioned user's home directory, if the user wants to keep the `~/` directory (`$HOME`) under version control.

This is the place where you can put patterns matching the backup or temporary files generated by your editor or IDE of choice.

Tip

Ignored files are considered expendable

Warning: Do not add precious files, that is those which you do not want to track in a given repository but whose contents are important, to the list of ignored files! The types of file that are ignored (excluded) by Git are either easy to re-generate (build products and other generated files), or not important to the user (temporary files, backup files).

Therefore Git considers ignored files expendable and will remove them without warning when required to do a requested command, for example, if the ignored file conflicts with the contents of the revision being checked out.

Listing ignored files

You can list untracked ignored files with the `--ignored` option to the status command:

```
$ git status --ignored
On branch master
Ignored files:
  (use "git add -f <file>..." to include in what will be
committed)

  rand.c~

no changes added to commit (use "git add" and/or "git commit -a")

$ git status --short --branch --ignored
## master
!! rand.c~
```

You could instead use the dry-run option of cleaning ignored files: `git clean -Xnd`, or the low-level (plumbing) command `git ls-files`:

```
$ git ls-files --others --ignored --exclude-standard  
rand.c~
```

The latter command can also be used to list tracked files that match ignore patterns. Having such files might mean that some files need to be un-tracked (perhaps because what was once a source file is now generated), or that ignore patterns are too broad. As Git uses the existence of a file in the staging area (cache) to know which files to track, this can be done with the following command:

```
$ git ls-files --cached --ignored --exclude-standard
```

Tip

Plumbing versus porcelain commands

Git commands are divided into two sets: high-level **porcelain** commands intended for interactive usage by the user, and low-level **plumbing** commands intended mainly for shell scripting. The major difference is that high-level commands have output that can change and that is constantly improving, for example, going from (no branch) to (detached from HEAD) in the `git branch` output in the detached from HEAD case; though some porcelain commands have the option (usually `--porcelain`) to switch to unchanging output. Their output and behavior are subject to configuration.

Another important difference is that plumbing commands try to guess what you meant, have default parameters, use the default configuration, and so on. Not so with plumbing commands. In particular you need to pass the `--exclude-standard` option to the `git ls-files` command to make it respect the default set of ignore files.

You can find more on this topic in [Chapter 10, Customizing and Extending Git](#).

Ignoring changes in tracked files

You might have files in the repository that are changed, but rarely committed. These can be various local configuration files that are edited to match the local setup, but should never be committed upstream. This can be a file containing the proposed name for a new release, to be committed when tagging the next released version.

You would want to keep such files in the *dirty* state most of the time, but you would like Git not to tell you about their changes all the time, in case you miss other changes because you are used to ignore such messages.

Git can be configured to skip checking the worktree (to assume that it is always up to date), and to use instead the staged version of the file, by setting the aptly named `skip-worktree` flag for a file. For this you would need to use the low-level `git update-index` command, the plumbing equivalent of the user-facing `git add` porcelain (you can check file status and flags with '`git ls-files`'):

```
$ git update-index --skip-worktree GIT-VERSION-NAME  
$ git ls-files -v  
S GIT-VERSION-NAME  
H Makefile
```

Note however that this elision of worktree also includes the `git stash` command; to stash away your changes and make the working directory clean, you need to disable this flag (at least temporarily). To make Git again look at the working directory version, and start tracking changes to the file, use the following command:

```
$ git update-index --no-assume-unchanged GIT-VERSION-NAME
```

Note

There is a similar `assume-unchanged` flag that can be used to make Git completely ignore any changes to the file, or rather assume that it is unchanged. Files marked with this flag never show as changed in the output of the `git status` or `git diff` command. The changes to it will not be staged nor committed.

This is sometimes useful when working with a big project on a filesystem with very slow checking for changes. *Do not use assume-unchanged for ignoring changes to tracked files.* You are promising that the file didn't change; lying to Git with, for example, `git stash save` believing what you stated, would lose your local changes.

File attributes

There are some settings and options in Git that can be specified on a per-path basis; similar to how ignoring files (marking files as intentionally untracked) works. These path-specific settings are called attributes.

To specify attributes for files matching a given pattern, you need to add a line with a pattern followed by a whitespace-separated list of attributes to one of the `gitattribute` files (similarly to how the `gitignore` files work):

- The per-user file, for attributes that should affect all repositories for a single user, specified by the configuration variable `core.attributesFile`, by default `~/.config/git/attributes`
- The per repository `.git/info/attributes` file in the administrative area of the local clone of the repository, for attributes that should affect only a single specific clone of the repository (for one user's workflow)
- The `.gitattributes` files in the working directories of a project, for those attributes that should be shared among developers

The rules for how patterns are used to match files are the same as for the `gitignore` files, described in an earlier section, except that there is no support for negative patterns.

Each attribute can be in one of the following states for a given path: *set* (special value `true`), *unset* (special value `false`), and set to given value, or *unspecified*:

```
pattern*  set -unset set-to=value !unspecified
```

Note that there can be no whitespace around the equals sign = when setting an attribute to a string value!

When more than one pattern matches the path, a later line overrides an earlier line on a per-attribute basis. `Gitattribute` files are used in order, from the per-user to the `.gitattributes` file in a given directory, like for `gitignore` files.

Identifying binary files and end-of-line conversions

Different operating systems and different applications can differ in how they represent newline in text files. Unix and Unix-like systems (including MacOS X) use a single control character LF (\n), while MS Windows uses CR followed by LF (\n\r); MacOS up to version 9 used CR alone (\r).

That might be a problem for developing portable applications if different developers use different operating systems. We don't want to have spurious changes because of different end of line conventions. Therefore Git makes it possible to automatically normalize **end of line (eol)** characters to be LF in the repository on commit (check-in), and optionally to convert them to *CR + LF* in the working directory on checkout.

You can control whether a file should be considered for end of line conversion with the `text` attribute. Setting it enables end-of-line conversion, unsetting it disables it. Setting it to the `auto` value makes Git guess if given file is a text file; if it is, end-of-line conversion is enabled. For files where the `text` attribute is unspecified, Git uses `core.autocrlf` to decide whether to treat them as `text=auto` case.

Tip

How Git detects if a file contains binary data

To decide whether a file contains binary data, Git examines the beginning of the file for an occurrence of a zero byte (the null character or \0). When deciding about converting a file (as in end-of-line conversion), the criterion is more strict: for a file to be considered text it must have no nulls, and no more than around 1% of it should be non-printable characters.

This means that Git usually considers files saved in the UTF-16 encoding

to be binary.

To decide what line ending type Git should use in the working directory for text files, you need to set up the `core.eol` configuration variable. This can be set to `crlf`, `lf`, or `native` (the last is the default). You can also force a specific line ending for a given file with the `eol=lf` or `eol=crlf` attribute.

Old crlf attribute	New text and eol attributes
<code>crlf</code>	<code>text</code>
<code>-crlf</code>	<code>-text</code>
<code>crlf=input</code>	<code>eol=lf</code>

Table 1. Backward compatibility with the crlf attribute

End of line conversion bears a slight chance of corrupting data. If you want Git to warn or prevent conversion for files with a mixture of LF and CRLF line endings, use the `core.safecrlf` configuration variable.

Sometimes Git might not detect that a file is binary correctly, or there may be some file that is nominally text, but which is opaque to a human reader. Examples include PostScript documents (`*.ps`) and Xcode build settings (`*.pbxproj`). Such files should be not normalized and textual diff for them doesn't make sense. You can mark such files explicitly as binary with the `binary` attribute macro (which is equivalent to `-text -diff`):

```
*.ps      binary  
*.pbxproj binary
```

Tip

Forcing end-of-line conversion when turning it on

When normalization of line endings is turned on in the repository (by editing the `.gitattributes` file) one should also force normalization of

files. Otherwise the change in newline representation will be misattributed to the next change to the file:

```
$ rm .git/index  
$ git reset  
$ git add -u  
$ git add .gitattributes
```

You can check which files will be normalized (for example, with `git status`) after `git reset` step, but before `git add -u`.

Diff and merge configuration

In Git, you can use the attributes functionality to configure how to show differences between different versions of a file, and how to do a 3-way merge of its contents. This can be used to enhance those operation, making `diff` more attractive and `merge` less likely to conflict. It can be even used to make it possible to effectively `diff` binary files.

In both cases we would usually need to set up the `diff` and/or `merge` driver. The attributes file only tells us which driver to use; the rest of the information is contained in the configuration file, and the configuration is not automatically shared among developers unlike the `.gitattributes` file (though you can create a shared configuration fragment, add it to the repository, and have developers include it in their local per-repository `config`, via the relative `include.path`). This is easy to understand—the tool configuration may be different on different computers, and some tools may be not available for the developer's operating system of choice. But this means that some information needs to be distributed out-of-band.

There are however a few built-in `diff` drivers and `merge` drivers that you can use.

Generating diffs and binary files

Generating diffs for particular files is affected by the `diff` attribute. If this attribute is unset, Git will treat files as binary with respect to generating diffs, and show just *binary files differ* (or a binary diff).

Setting it will force Git to treat a file as text, even if it contains byte sequences that normally mark the file as binary, such as the null (\0) character.

You can use the `diff` attribute to make Git effectively describe the differences between two versions of a binary file. In this you have two options: the easier one is to tell Git how to convert a binary file to a text format, or how to extract text information (for example metadata) from binary data. This text representation is then compared using the ordinary textual diff command. Even though conversion to text usually loses some information, the resulting diff is useful for human viewing (even though it is not information about all the changes).

This can be done with the `textconv` config key for a diff driver, where you specify a program that takes the name of the file as an argument and returns a text representation on its output.

For example, you might want to see the diff of the contents of MS Word documents, and see the difference in metadata for JPEG images. First you need to put something like this in your `.gitattributes` file:

```
*.doc  diff=word
*.jpg  diff=exif
```

You can for example use the `catdoc` program to extract text from binary MS Word documents, and the `exiftool` to extract EXIF metadata from JPEG images. Because conversion can be slow, Git provides a mechanism to cache the output in the form of the Boolean `cachetextconv` attribute; the cached data is stored using *notes* (this mechanism will be explained in [Chapter 8, Keeping History Clean](#)). The part of the configuration file responsible for this setup looks like this:

```
[diff "word"]
textconv = catdoc

# cached data stored in refs/notes/textconv/exif
[diff "exif"]
textconv = exiftool
cachetextconv = true
```

You can see how the output of the `textconv` filter looks with `git show` or `git cat-file -p` with the `--textconv` option.

The more complicated but also more powerful option is to use an *external diff driver* (an attribute version of the global driver that can be specified with the `GIT_EXTERNAL_DIFF` environment variable or the `diff.external` configuration variable) with the `command` option of the diff driver. On the other hand, you lose some options that Git diff gives: colorization, word diff, and combined diff for merges.

Such a program will be called with seven parameters: `path`, `old-file`, `old-hex`, `old-mode`, `new-file`, `new-hex`, and `new-mode`. Here `old-file` and `new-file` are files that the diff driver can use to read the contents of two versions of the differing file, `old-hex` and `new-hex` are SHA-1 identifiers of file contents, and `old-mode` and `new-mode` are octal representations of file modes. The command is expected to generate diff-like output. For example, you might want to use the XML-aware diff tool to compare XML files:

```
$ echo "*.*xml diff=xmldiff" >>.gitattributes  
$ git config diff.xmldiff.command xmldiff-wrapper.sh
```

This example assumes that you have written the `xmldiff-wrapper.sh` shell script to reorder options to fit the XML diff tool.

Configuring diff output

The diff format that Git uses to show changes for users was described in detail in [Chapter 3, *Developing with Git*](#). Each group of changes (called a **chunk**) in textual diff output is preceded by the chunk header line, for example:

```
@@ -18,6 +19,8 @@ int cmd_http_fetch(int argc, const char **argv,
```

The text after the second `@@` is meant to describe the section of file where the chunk is; for C source files it is the start of the function. Decision on how to detect the description of such a section depends of course on the type of file. Git allows you to configure this by setting the

`xfuncname` configuration option of the `diff` driver to the regular expression which match the description of the section of the file. For example, for LaTeX documents you might want to use the following configuration for the `tex` diff driver (but you don't need to, as `tex` is one of the pre-defined, built-in diff drivers).

```
[diff "tex"]
xfuncname = "^\\\\\\((sub)*section\\{.*}\\)$"
wordRegex = "\\\\([a-zA-Z]+|[{}]|\\\\.|[{}[:space:]]+|
```

The `wordRegex` configuration defines what word is in LaTeX documents for `git diff --word-diff`.

Performing a 3-way merge

You can also use the `merge` attribute to tell Git to use specific merge strategies for specific files or classes of files in your project. Git by default will use the 3-way merge driver (similar to `rcsmerge`) for text files, and it will take our (being merged) version and mark the result as a conflicted merge for binary files. You can force a 3-way merge by setting the `merge` attribute (or by using `merge=text`); you can force binary-like merging by unsetting this attribute (with `-merge`, which is equivalent to `merge=binary`).

You can also write your own merge driver, or configure Git to use a third-party external merge driver. For example, if you keep a GNU-style `ChangeLog` file in your repository (with a curated list of changes with their description), you can use the `git-merge-changelog` command from the GNU Portability Library (Gnulib). You need to add the following to the appropriate Git config file:

```
[merge "merge-changelog"]
  name = GNU-style ChangeLog merge driver
  driver = git-merge-changelog %O %A %B
```

Here the token `%O` in `merge.merge-changelog.driver` will be expanded to the name of the temporary file holding the contents of the merge ancestor's (old) version. Tokens `%A` and `%B` expand to the names of temporary files holding contents being merged, respectively the current

(ours, merged into) version and the other branches' (theirs, merged) version. The merge driver is expected to leave the merged version in the `%A` file, exiting with non-zero status if there is a merge conflict.

Note that you can use a different driver for an internal merge between common ancestors (when there is more than one). This is done with `merge.*.recursive`—for example using the predefined `binary` driver.

Of course you will also need to tell Git to use this driver for `ChangeLog` files, adding the following line to `.gitattributes`:

```
ChangeLog merge=merge-changelog
```

Transforming files (content filtering)

Sometimes you might want to massage the content into a shape that is more convenient for Git, the platform (operating system), the file system, and the user to use. End of line conversion can be considered a special case for such an operation.

To do this, you need to set the `filter` attribute for appropriate paths, and to configure the `clean` and `smudge` commands of specified filter driver (either command can be left unspecified for a pass-through filter). When checking out the file matching given pattern, the `smudge` command is fed file contents from the repository in its standard input, and its standard output is used to update the file in the working directory:

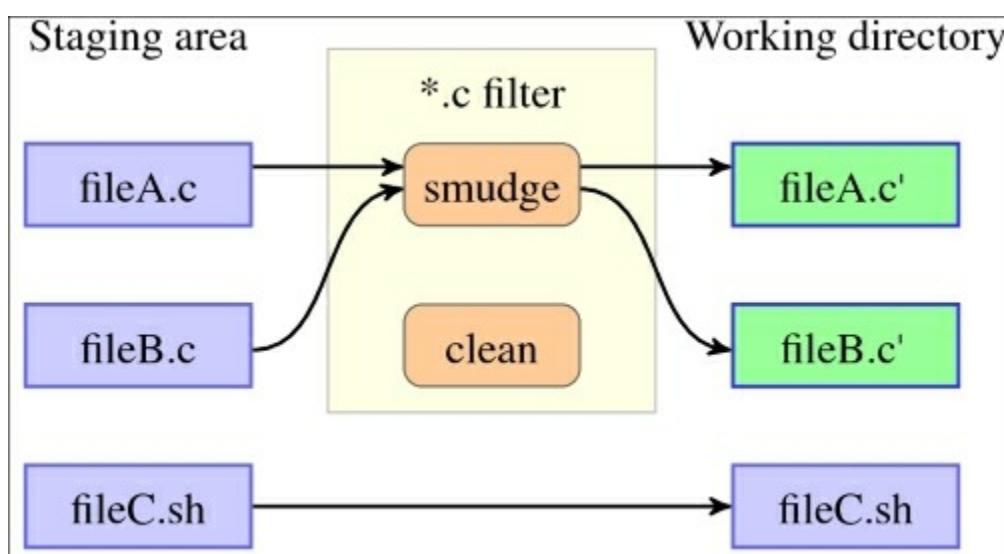


Fig 1. The "smudge" filter is run on checkout (when writing files to the working directory).

Similarly, the `clean` command of a filter is used to convert the contents of the worktree file to a shape suitable to be stored in the repository:

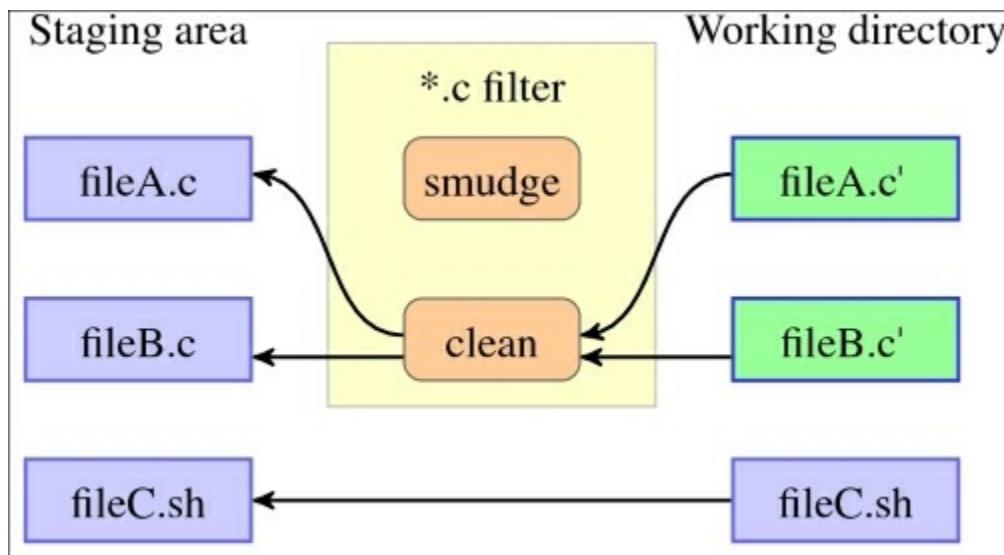


Fig 2. The "clean" filter is run when files are staged (added to the index—the staging area).

When specifying a command, you can use the `%f` token, which will be replaced by the name of the file the filter is working on.

One simple example is to use `rezip` script for **OpenDocument Format (ODF)** files. ODF documents are ZIP archives of mainly XML files. Git uses compression itself and also does deltification (but cannot do it on already compressed files); the idea is to store uncompressed files in the repository, but to checkout compressed files:

```
[filter "opendocument"]
  clean = "rezip -p ODF_UNCOMPRESS"
  smudge = "rezip -p ODF_COMPRESS"
```

Of course you also need to tell Git to use this filter for all kinds of ODF

files:

```
*.odt filter=opendocument  
*.ods filter=opendocument  
*.odp filter=opendocument
```

Another example of an *advisory* filter is to use the `indent` program to force a code formatting convention; a similar example would be to replace tabs with spaces on check-in:

```
[filter "indent"]  
    clean = indent
```

Obligatory file transformations

Another use of content filtering is to store the content that cannot be directly used in the repository and turn it into a usable form upon checkout.

One such example might be use `gitattributes` to store large binary files, used only by a subset of developers, outside the Git repository; inside the repository there is only an identifier that allows us to get file contents from external storage. That's how `git-media` works:

```
$ git config filter.media.clean "git-media filter-clean"  
$ git config filter.media.smudge "git-media filter-smudge"  
$ echo "*.* filter=media -crlf" >> .gitattributes
```

Note

You can find the `git-media` tool at <https://github.com/schacon/git-media>. Other similar tools will be mentioned in [Chapter 9, Managing Subprojects - Building a Living Framework](#), as alternative solutions to the problem of handling large files.

Another example would be encrypting sensitive content, or replacing a local sensitive program configuration that is required for an application to work (for example, a database password) with a placeholder. Because running such a filter is required to get useful contents, you can mark it as such:

```
[filter "clean-password"]
  clean = sed -e 's/^pass = .*$/pass = @PASSWORD@/'
  smudge = sed -e 's/^pass = @PASSWORD@/pass = passw0rd/'
  required
```

Note that this is only a simplified example; in real use you would have to consider the security of the `config` file itself if you do this, or store the real password in an external smudge script. In such case you'd better also set up a `pre-commit`, `pre-push`, and `update` hook to ensure that the password won't make it to the public repository (see [Chapter 10, *Customizing and Extending Git*](#) for details).

Keyword expansion and substitution

Sometimes there is a need to have a piece of dynamic information about the versioned file in the contents of the file itself. To keep such information up to date you can request the version control system to do the *keyword expansion*: replace the *keyword anchor* in the form of a string of text (in the file contents) formatted like the following: `$Keyword$`, with the keyword inside dollar characters (keyword anchor), which is usually replaced by VCS with `$Keyword: value$`, that is keyword followed by its expansion.

The main problem with this in Git is that you cannot modify the file contents stored in the repository with information about the commit after you've committed because of the way Git works (more information about this can be found in [Chapter 8, *Keeping History Clean*](#)). This means that keyword anchors must be stored in the repository as-is, and only expanded in the worktree on checkout. However, this is also an advantage; you would get no spurious differences due to keyword expansion when examining the history.

The only built-in keyword that Git supports is `Id`: its value is the SHA-1 identifier of the file contents (the SHA-1 checksum of the blob object representing the file contents, which is not the same as the SHA-1 of the file; see [Chapter 8, *Keeping History Clean*](#), for how objects are constructed). You need to request this keyword expansion by setting the `ident` attribute for a file.

You can however write your own keyword expansion support with an appropriate filter, defining the `smudge` command that would expand the keyword, and the `clean` command that would replace the expanded keyword with its keyword anchor.

With this mechanism you can, for example, implement support for the `$Date$` keyword, expanding it on checkout to the date when the file was last modified:

```
[filter "dater"]
  clean = sed -e 's/\\\\\\\$Date[^\\\\\\\$]*\\\\\\$/\\\\\\\$Date\\\\\\$/'
  smudge = expand_date %f
```

The `expand_date` script, which is passed the name of file as an argument, could for example run the `git log --pretty=format:"%ad" "$1"` command to get the substitution value.

You need however to remember another limitation. Namely, for a better performance, Git does not touch files that did not change, be it on commit, on switching the branch (on checkout), or on rewinding the branch (on reset). This means that this trick cannot support the keyword expansion for date of the last revision of a project (as opposed to the last revision that changed the file).

If you need to have such information in distributed sources (for example, the description of the current commit, how long since the tagged release), you can either make it a part of build system, or use *keyword substitution* for the `git archive` command. The latter is quite a generic feature: if the `export-subst` attribute is set for a file, Git will expand the `$Format:<PLACEHOLDERS>$` generalized keyword when adding the file to an archive.

Note

The expansion of the `$Format$` meta-keyword depends on the availability of the revision identifier; it cannot be done if you, for example, pass the SHA-1 identifier of a tree object to the `git archive` command.

The placeholders are the same as for the `--pretty=format:` custom formats for `git log`, which are described in [Chapter 2, Exploring Project History](#). For example, the string `$Format:%H$` will be *replaced* (not expanded) by the commit hash. It is an irreversible keyword substitution; there is no trace of the keyword in the result of the archive (export) operation.

Other built-in attributes

You can also tell Git not to add certain files or directories when generating an archive. For example, you might want to not include in the user-facing archive the directory with distribution tests, which are useful for the developer but not for end users (they may require additional tools, or check the quality of the program and process rather than the correctness of the application behavior). This can be done by setting the `export-ignore` attribute, for example, by adding the following line to `.gitattributes`:

```
xt/ export-ignore
```

Another thing that can be configured with file attributes is defining what `diff` and `apply` should consider a whitespace error for specific types of file; this is a fine-grained version of the `core.whitespace` configuration variable. Note that the list of common whitespace problems to take notice of should use commas as an element separator, without any surrounding whitespace, when put in the `.gitattributes` file. See the following example (taken from the Git project):

```
* whitespace=!indent,trail,space
*.ch whitespace=indent,trail,space
*.sh whitespace=indent,trail,space
```

With file attributes you can also specify the character encoding that is used by a particular file, by providing it as a value of the `encoding` attribute. Git can use it to select how to display the file in GUI tools (for example, `gitk` and `git gui`). This is a fine-grained version of the `gui.encoding` configuration variable, and is used only when explicitly asked for due to performance considerations. For example, GNU gettext

Portable Object (.po) files holding translations should use the UTF-8 encoding:

```
/po/* .po      encoding=UTF-8
```

Defining attribute macros

In the *Identifying binary files and end-of-line conversions* section of this chapter, we learned to mark binary files with the `binary` attribute. This is actually the attribute macro expanding to `-diff -merge -text` (unsetting three file attributes). It would be nice to define such macros to avoid unnecessary duplication; there can be more than one pattern matching given type of files, but one `gitattribute` line can contain only one file pattern. Git allows defining such macros, but only in top-level `gitattributes` files: `core.attributesFile`, `.git/info/attributes`, or `.gitattributes` in the main (top level) directory of a project. The built-in `binary` macro could have been defined as follows:

```
[attr]binary -diff -merge -text
```

You can also define your own attributes. You can then programmatically check which attributes are set for a given file, or what the value is of an attribute for a set of files, with the `git check-attr` command.

Fixing mistakes with the reset command

At any stage during development, you might want to undo something, to fix mistakes, or to abandon the current work. There is no `git undo` command in core Git, and neither is there support for the `--undo` option in Git commands, though many commands have an `--abort` option to abandon current work in progress. One of the reasons why there is no such command or option yet is the ambiguity on what needs to be undone (especially for multi-step operations).

Many mistakes can be fixed with the help of the `git reset` command. It can be used for various purposes and in various ways; understanding how this command works will help you in using it in many situations, not limited to provided example usage.

Note that this section covers only the full-tree mode of `git reset`; resetting the state of a file, that is the description of what `git reset --<file>` does, is left for the *Managing worktree and staging area* section at the end of this chapter.

Rewinding the branch head, softly

The `git reset` command in its full-tree mode affects the current branch head, and can also affect the index (the staging area) and the working directory. Note that `reset` does not change which branch is current, as opposed to `checkout`—the difference is described in [Chapter 3, *Developing with Git*](#).

To reset only the current branch head, and not touch the index or the working tree, you use `git reset --soft [<revision>]`.

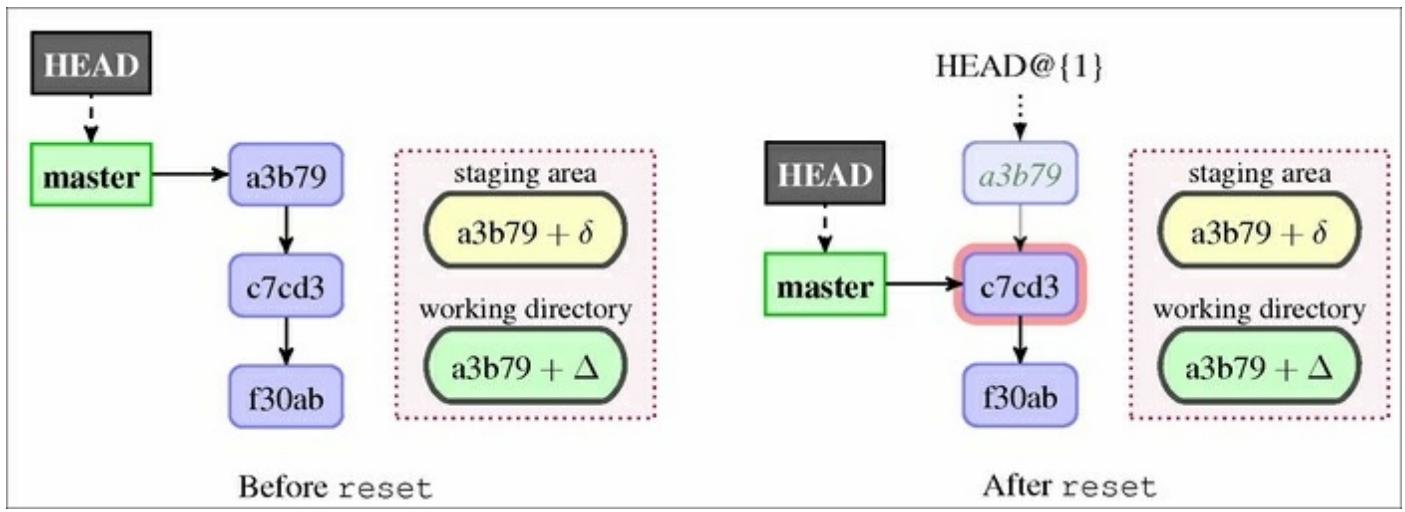


Fig 3. Before and after soft reset

Effectively, we are just changing the pointer of the current branch (`master` in the example shown in *Fig 3*) to point to a given revision (`HEAD^`— the previous commit in the example). Neither the staging area nor the working directory are affected. This leaves all your changed files (and all files that differ between the old and new revision pointed by branch) in the *Changes to be committed* state, as `git status` would put it.

Removing or amending a commit

The way the command works means that a *soft reset* can be used to undo the act of creating a commit. This can be used for example to amend a commit, though it is far easier to simply use the `--amend` option of `git commit`. In fact, running the following command:

```
$ git commit --amend [<options>]
```

is equivalent to:

```
$ git reset --soft HEAD^
$ git commit --reedit-message=ORIG_HEAD [<options>]
```

The `git commit --amend` command also works for merge commits as opposed to using soft reset. When amending commit, if you want to just

fix the commit message there will be no additional options. If you want to include a fix from the working directory without changing the commit message, you can use `-a --no-edit`. If you want to fix the authorship information after correcting Git configuration, use `--reset-author --no-edit`.

Squashing commits with reset

You are not limited to rewinding the branch head to just the previous commit. Using a soft reset, you can squash a few earlier commits (for example, `commit` and `bugfix`, or introducing new functionality and using it), making one commit out of two (or more); alternatively, you can instead use the `squash` instruction of **interactive rebase**, as described in [Chapter 8, Keeping History Clean](#). With the latter, you can actually squash any series of commits, not just most recent ones.

Resetting the branch head and the index

The default mode of `reset` command, so called **mixed reset** (because it is between the soft and hard forms), changes the current branch head to point to a given revision, and also resets the index, putting the contents of that revision into the staging area:

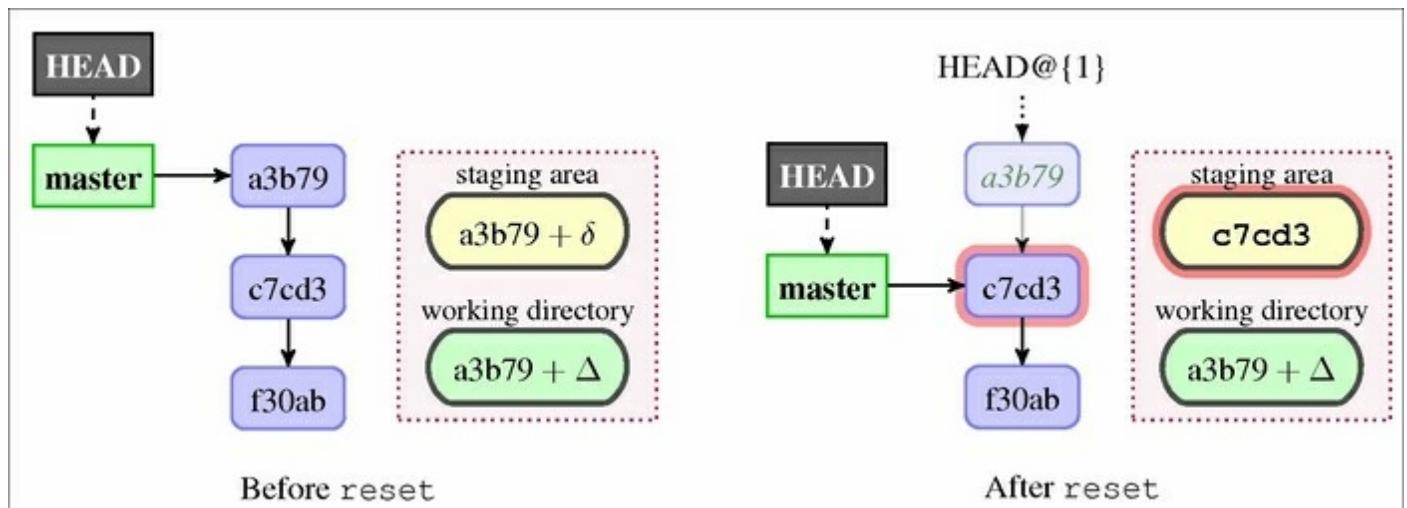


Fig 4. Before and after mixed reset

This leaves all your changed files (and all files that differ between the

old and new revision pointed by branch) in the *Changes not staged for commit* state, as `git status` would put it. The `git reset --mixed` command will report what has not been updated, using the short status format.

This version of reset command can be used, for example, to undo all additions of new files. This can be done by running `git reset`, assuming that you didn't stage any changes (or that you can put up with losing them). If you want to un-add a particular file, use `git rm --cached <file>`.

Splitting a commit with reset

You can use a mixed reset to split a commit in two. First, run `git reset HEAD^` to reset the branch head and the index to the previous revision. Then interactively add changes that you want to have in the first commit, and then create this first commit from the index (`git add -i` and `git commit`). A second commit can then be created from the working directory state (`git commit -a`).

If it is easier to interactively remove changes, you can do this too. Use `git reset --soft HEAD^`, interactively un-stage changes with an interactive per-file reset, create the first commit from the constructed state in the index, and create the second commit from the working directory.

Here again you can instead use the interactive rebase to split commits further in the history. The rebase operation will switch to the appropriate commit, and the actual splitting would probably be done as described here.

Saving and restoring state with the WIP commit

Suppose you are interrupted by an urgent fix request while you are in the middle of work on the development branch. You don't want to lose your changes, but the worktree is a bit of a mess. One possible solution is to save the current state of the working area by creating a temporary commit:

```
$ git commit -a -m 'snapshot WIP (Work In Progress)'
```

Then you handle the interruption, switching to the maintenance branch and creating a commit to fix the issue. Then you need to go back to the previous branch (by using checkout), remove the `WIP` commit from the history (using a soft reset), and go back to the un-staged starting state (with a mixed reset):

```
$ git checkout -
$ git reset --soft HEAD^
$ git reset
```

Though it is much easier to just use `git stash` instead to handle interruptions, see the *Stashing away your changes* section in this chapter. On the other hand, such temporary commits (or similar proof-of-concept work) can be shared with other developers, as opposed to stash.

Discarding changes and rewinding branch

Sometimes your files will get in such a mess that you want to discard all changes, and to return the working directory and the staging area (the index) to the last committed state (the last good version). Or you might want to rewind the state of the repository to an earlier version. A *hard reset* will change the current branch head and reset the index and the working tree. Any changes to tracked files are discarded.

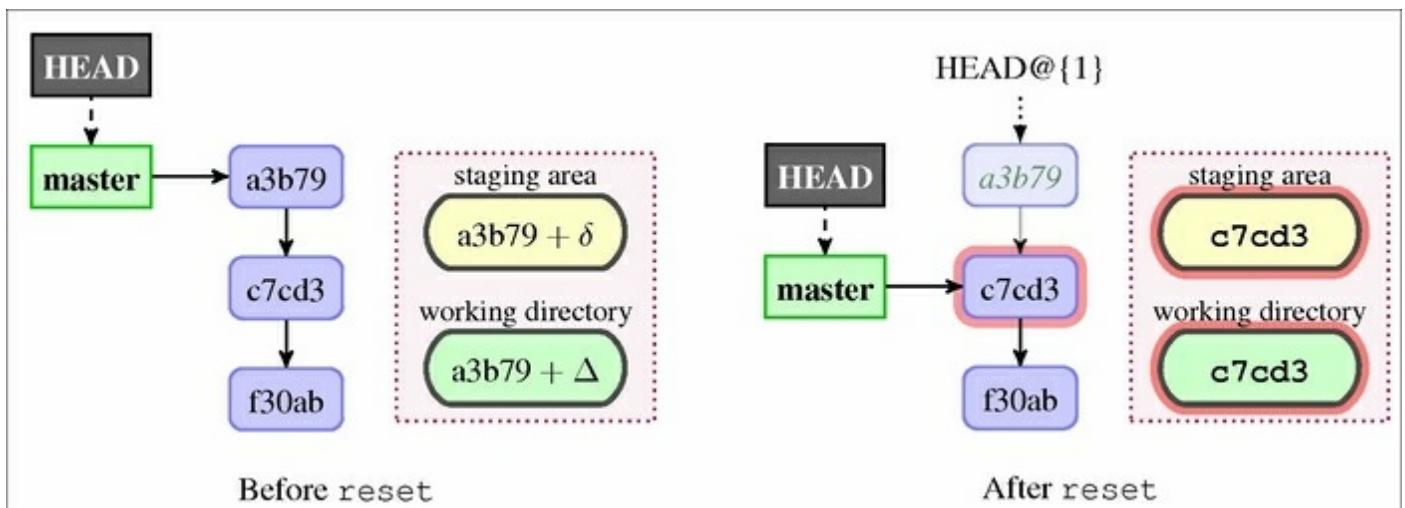


Fig 5. Before and after hard reset

This command can be used to undo (remove) a commit as if it had never happened. Running `git reset --hard HEAD^` will effectively discard the last commit (though it will be available for a limited time via reflog), unless it is reachable from some other branch.

Another common usage is to discard changes to the working directory with `git reset --hard`.

Note

It is very important to remember that a hard reset would irrecoverably remove all changes from the staging area and working directory. You cannot undo this part of the operation! Changes are lost forever!

Moving commits to a feature branch

Say that you were working on something on the `master` branch, and you have already created a sequence of commits. You realize that the feature you are working on is more involved, and you want to continue polishing it on a topic branch, as described in [Chapter 6, Advanced Branching Techniques](#). You want to move all those commits that are in `master` (let's say, the last three revisions) to the aforementioned feature branch.

You need to create the feature branch, save uncommitted changes (if any), rewind `master` removing those *topical* commits from it, and finally switch to the feature branch to continue working (or you can use rebase instead):

```
$ git branch feature/topic
$ git stash
No local changes to save
$ git reset --hard HEAD~3
HEAD is now at f82887f before
$ git checkout feature/topic
Switched to branch 'feature/topic'
```

Of course, if there were local changes to save, this preceding series of

commands would have to be followed by `git stash pop`.

Undoing a merge or a pull

Hard resets can also be used to abort a failed merge with `git reset --hard HEAD` (the `HEAD` is the default value for revision and can be omitted), for example, if you decide that you don't want to resolve the merge conflict at this time (though with modern Git you can use `git merge --abort` instead).

You can also remove a successful fast-forward pull or undo a rebase (and many other operations moving the branch head) with `git reset --hard ORIG_HEAD`. (You can here use `HEAD@{1}` instead of `ORIG_HEAD`).

Safer reset – keeping your changes

A hard reset will discard your local changes, similarly to the way `git checkout -f` would. Sometimes you might want to rewind the current branch while keeping local changes: that's what `git reset --keep` is for.

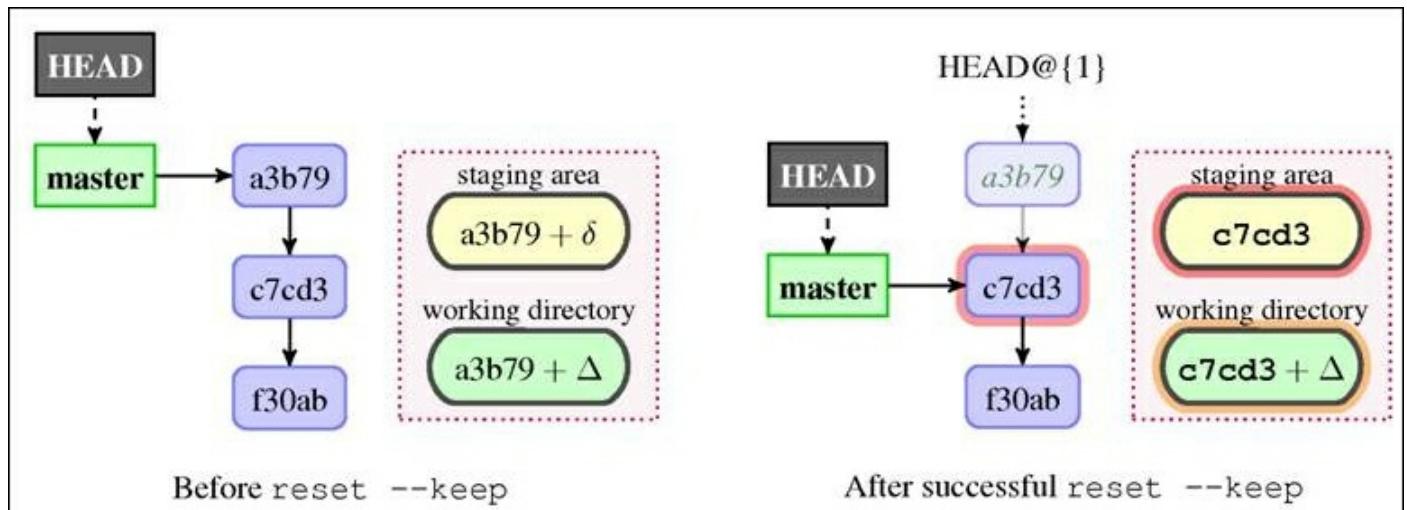


Fig 6. Before and after successful `git reset --keep HEAD^`.

This mode resets the staging area (index entries), but retains the unstaged (local) changes that are currently in the working directory. If it is not possible, the reset is aborted. This means that local changes in the worktree are preserved and moved to the new commit, in a similar way

to how `git checkout <branch>` works with uncommitted changes. The successful case is a bit like stashing changes away, hard resetting, then unstashing.

Note

The way `git reset --keep <revision>` works is by updating the version (in the working directory) of only those files that are different between the revision we rewind to and the `HEAD`. The reset is aborted if there is any file that is different between `HEAD` and `<revision>` (and thus should be updated) and has local uncommitted changes.

Rebase changes to an earlier revision

Suppose that you are working on something, but now you realize that what you have in your working directory should be in another branch, unrelated to a previous commit. For example, you might have started to work on a bug while on the `master` branch, and only then realized that it also affects the maintenance branch `maint`. This means that the fix should therefore be put earlier in a branch, starting from the common ancestor of those branches (or a place where the bug was introduced). This would make it possible to merge the same fix both into `master` and `maint`, as described in [Chapter 12, Git Best Practices](#):

```
$ edit  
$ git checkout -b bugfix-127  
$ git reset --keep start
```

An alternate solution would be to simply use `git stash`:

```
$ edit  
$ git stash  
$ git checkout -b bugfix-127 start  
$ git stash pop
```

Stashing away your changes

Often, when you've been working on a project, and things are in a messy state not suitable for a permanent conflict, you want to temporarily save the current state and go to work on something else. The answer to this problem is the `git stash` command.

Stashing takes the dirty state of your working area—that is, your modified tracked files in your worktree (though you can also stash untracked files with the `--include-untracked` option), and the state of the staging area, then saves this state, and resets both the working directory and the index to the last committed version (to match the `HEAD` commit), effectively running `git reset --hard HEAD`. You can then reapply the stashed changes at any time.

Stashes are saved on a stack: by default you apply the last stashed changes (`stash@{0}`), though you can list stashed changes (with `git stash list`), and explicitly select any of the stashes.

Using git stash

If you don't expect for the interruption to last long, you can simply stash away your changes, handle the interruption, then unstash them:

```
$ git stash  
$ ... handle interruption ...  
$ git stash pop
```

By default `git stash pop` will apply the last stashed changes, and delete the stash if applied successfully. To see what stashes you have stored, you can use `git stash list`:

```
$ git stash list  
stash@{0}: WIP on master: 049d078 Use strtol(), atoi() is  
deprecated  
stash@{1}: WIP on master: c264051 Error checking for <number>
```

You can use any of the older stashes by specifying the stash name as an

argument. For example, you can run `git stash apply stash@{1}` to apply it, and you can drop it (remove it from the list of stashes) with `git stash drop stash@{1}`; the `git stash pop` command is just a shortcut for `apply + drop`.

The default description that Git gives to a stash (*WIP on branch*) is useful for remembering where you were when stashing the changes (giving branch and commit), but doesn't help you remember what you were working on, and what is stashed away. However, you can examine the changes recorded in the stash as a diff with `git stash show -p`. But if you expect that the interruption might be more involved, you should better save the current state to a stash with a description of what you were working on:

```
$ git stash save 'Add <count>'  
Saved working directory and index state On master: Add <count>  
HEAD is now at 049d078 Use strtol(), atoi() is deprecated
```

Git would then use the provided message to describe stashed changes:

```
$ git stash list  
stash@{0}: On master: Add <count>  
stash@{1}: WIP on master: c264051 Error checking for <number>
```

Sometimes the branch you were working on when you ran `git stash save` has changed enough that `git stash pop` fails, because there are new revisions past the commit you were on when stashing the changes. If you want to create a regular commit out of the stashed changes, or just test stashed changes, you can use `git stash branch <branch name>`. This will create a new branch at the revision you were at when saving the changes, switch to this branch, reapply your work there, and drop stashed changes.

Stash and the staging area

By default, stashing resets both the working directory and the staging area to the `HEAD` version. You can make `git stash` keep the state of the index, and reset the working area to the staged state, with the `--keep-index` option.

This is very useful if you used the staging area to untangle changes in the working directory, as described in the section about interactive commits in [Chapter 3](#), *Developing with Git*, or if you want to split the commit in two as described in *Splitting a commit with reset* section in this chapter. In both cases you would want to test each change before committing. The workflow would look like the following:

```
$ git add --interactive  
$ git stash --keep-index  
$ make test  
$ git commit -m 'First part'  
$ git stash pop
```

You can also use `git stash --patch` to select how the working area should look after stashing away the changes.

When restoring stashed changes, Git will ordinarily try to apply only saved worktree changes, adding them to the current state of the working directory (which must match the staging area). If there are conflicts while applying the state, they are stored in the index as usual—Git won't drop the stash if there were conflicts.

You can also try to restore the saved state of the staging area with the `--index` option; this will fail if there are conflicts when applying working tree changes (because there is no place to store conflicts; the staging area is busy).

Stash internals

Perhaps you applied stashed changes, did some work, and then for some reason want to un-apply those changes that originally came from the stash. Or you have mistakenly dropped the stash, or cleared all stashes (which you can do with `git stash clear`), and would like to recover them. Or perhaps you want to see how the file looked when you stashed away changes. For this, you need to know what Git does when creating a stash.

To stash away your changes, Git creates two automatic commits: one for

the index (staging area), and one for the working directory. With `git stash --include-untracked`, Git creates an additional third automatic commit for untracked files.

The commit containing the work in progress in the working directory (the state of files tracked from there) is the stash, and has the commit with the contents of the staging area as its second parent. This commit is stored in a special ref: `refs/stash`. Both WIP (stash) and index commits have the revision you were on when saving changes as its first (and only for the `index` commit) parent.

We can see this with `git log --graph` or `gitk`:

```
$ git stash save --quiet 'Add <count>'  
$ git log --oneline --graph --decorate --boundary stash ^HEAD  
* 81ef667 (refs/stash) On master: Add <count>  
|\  
| * ed95050 index on master: 765b095 Added .gitignore  
|/  
o 765b095 (HEAD, master) Added .gitignore  
$ git show-ref --abbrev  
765b095 refs/heads/master  
81ef667 refs/stash
```

We had to use `git show-ref` here (we could have used `git for-each-ref` instead), because `git branch -a` shows only branches, not arbitrary refs.

When saving untracked changes, the situation is similar:

```
$ git stash --include-untracked  
Saved working directory and index state WIP on master: 765b095  
Added\  
.gitignore  
HEAD is now at 765b095 Added .gitignore  
$ git log --oneline --graph --decorate --boundary stash ^HEAD  
*-.. bb76632 (refs/stash) WIP on master: 765b095 Added  
.gitignore  
|\ \  
| | * 1ae1716 untracked files on master: 765b095 Added  
.gitignore  
| * d093b52 index on master: 765b095 Added .gitignore
```

```
1/  
o 765b095 (HEAD, B) Added .gitignore
```

We see that the untracked file commit is the third parent of the WIP commit, and that it doesn't have any parents.

Well, that's how stashing works, but how does Git maintain the stack of stashes? If you have noticed that the `git stash list` output and the `stash@{<n>}` notation therein looks like reflog, you have guessed right; Git finds older stashes in the reflog for the `refs/stash` reference:

```
$ git reflog stash  
81ef667 stash@{0}: On master: Add <count>  
bb76632 stash@{1}: WIP on master: Added .gitignore
```

Un-applying a stash

Let's take the first example from the beginning of the section: un-applying changes from the earlier `git stash apply`. One possible solution to achieve the required effect is to retrieve the patch associated with working directory changes from a stash, and apply it in reverse:

```
$ git stash show -p stash@{0} | git apply -R -
```

Note the `-p` option to the `git stash show` command—it forces patch output instead of a summary of changes. We could use `git show -m stash@{0}` (the `-m` option is necessary because a WIP commit representing the stash is a merge commit), or even simply `git diff stash@{0}^1 stash@{0}`, in place of `git stash show -p`.

Recovering stashes that were dropped erroneously

Let's try the second example: recovering stashes that were accidentally dropped or cleared. If they are still in your repository, you can search all commit objects that are unreachable from other refs and look like stashes (that is, they are merge commits and have a commit message using a strict pattern).

A simplified solution might look like this:

```
$ git fsck --unreachable |  
grep "unreachable commit " | cut -d" " -f3 |  
git log --stdin --merges --no-walk --grep="WIP on "
```

The first line finds all unreachable (lost) objects, the second one filters out everything but commits and extracts their SHA-1 identifiers, and third line filters out even more, showing only merge commits with a commit message containing the "WIP on " string.

This solution would not, however, find stashes with a custom message (those created with `git stash save "message"`).

Managing worktrees and the staging area

In [Chapter 3, *Developing with Git*](#), we learned that, besides the working directory where you work on changes, and the local repository where you store those changes as revisions, there is also a third section between them: the staging area, sometimes called the index.

In the same chapter, we also learned how to examine the status of the working directory, and how to view the differences. We saw how to create a new commit out of the working directory, or out of the staging area.

Now it is time to learn how to examine and modify the state of individual files.

Examining files and directories

It is easy to examine the contents of the working directory: just use the standard tools for viewing files (for example, an editor or a pager) and examining directories (for example, a file manager or the `dir` command). But how do we view the staged contents of a file, or the last committed version?

One possible solution is to use the `git show` command with the appropriate selector. [Chapter 2, *Exploring Project History*](#), gave us the `<revision>:<pathname>` syntax to examine the contents of a file at a given revision. Similar syntax can be used to retrieve the staged contents, namely `:<pathname>` (or `:<stage>:<pathname>` if the file is in a merge conflict; `:<pathname>` on itself is equivalent to `:0:<pathname>`).

Let's assume that we are in the `src/` subdirectory, and want to see the contents of the `rand.c` file there as it is in the working directory, in the staging area (using the absolute and relative path), and in the last commit:

```
src $ less -FRX rand.c
src $ git show :src/rand.c
src $ git show ./rand.c
src $ git show HEAD:src/rand.c
src $ git show HEAD:./rand.c
```

To see what files are staged in the index, there is the `git ls-files` command. By default it operates on the staging area contents, but can also be used to examine the working directory (which, as we have seen in this chapter, can be used to list ignored files). This command lists all files in the specified directory, or the current directory (because the index is a flat list of files, similar to `MANIFEST` files); you can use `:/` to denote the top-level directory of a project. Without using the `--full-name` option, it would show filenames relative to the current directory (or the one specified as parameter). In all examples it is assumed that we are in the `src/` subdirectory, as seen in command prompt.

```
src $ git ls-files
rand.c
src $ git ls-files --full-name :/
COPYRIGHT
Makefile
README
src/rand.c
```

What about committed changes? How can we examine which files were in a given revision? Here `git ls-tree` comes to the rescue (note that it is a plumbing command and does not default to the `HEAD` revision):

```
src $ git ls-tree --name-only HEAD
rand.c
src $ git ls-tree --abbrev --full-tree -r -t HEAD
100644 blob 862aafd      COPYRIGHT
100644 blob 25c3d1b      Makefile
100644 blob bdf2c76      README
040000 tree 7e44d2e      src
100644 blob b2c087f      src/rand.c
```

Searching file contents

Let's assume that you were reviewing code in the project and noticed an erroneous doubled semicolon ';;' in the C source code. Or perhaps you

were editing the file and noticed a bug nearby. You fix it, but you wonder: "How many of those mistakes are there?"—you would like to create a commit to fix every and each such errors.

Or perhaps you want to search the version scheduled for the next commit? Or maybe examine how it looks in the `next` branch?

With Git, you can use the `git grep` command:

```
$ git grep -e ';;'
```

This will only search tracked files in the working directory, from the current directory downwards. We will get many false positives, for example, from shell scripts—let's limit the search space to C source files:

```
$ git grep -e ';;' -- '*.c'
```

The quotes are necessary for Git to do expansion (path limiting), instead of `git grep` getting the list of files expanded by the shell. We still have many false matches from the *forever loop* C idiom:

```
for (;;) {
```

With `git grep` you can construct complex conditions, excluding false positives. Say that we want to search the whole project, not only the current directory:

```
$ git grep -e ';;' --and --not 'for *(*;;' -- '**/*.c'
```

To search the staging area, use `git grep --cached` (or the equivalent, and perhaps easier to remember, `git grep --staged`). To search the `next` branch, use `git grep next --`; similar command can be used to search any version, actually.

Un-tracking, un-staging, and un-modifying files

If you want to undo some file-level operation (if for example you have changed your mind about tracking files, or about staging changes)—look

no further than `git status` hints:

```
$ git status --ignored
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

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)

Untracked files:
  (use "git add <file>..." to include in what will be committed)

Ignored files:
  (use "git add -f <file>..." to include in what will be
committed)
```

You need to remember that only the contents of the working directory and the staging area can be changed. Committed changes are immutable.

If you want to undo adding a previously untracked file to the index—or remove a formerly tracked file from the staging area so that it would be *deleted* (not present) in the next commit, while keeping it in the working directory—use `git rm --cached <file>`.

Tip

Difference between the `--cached` (`--staged`) and `--index` options

Many Git commands, among others `git diff`, `git grep`, and `git rm`, support the `--cached` option (or its alias `--staged`). Others, such as `git stash`, have the `--index` option (the index is an alternate name for the staging area). These are *not* synonyms (as we will later see with `git apply` command, which supports both).

The `--cached` option is used to ask the command that usually works on files in the working directory to *only* work on the staged contents *instead*. For example, `git grep --cached` will search the staging area

instead of the working directory, and `git rm --cached` will only remove a file from the index, leaving it in the worktree.

The `--index` option is used to ask the command that usually works on files in the working directory to *also* affect the index, *additionally*. For example, `git stash apply --index` not only restores stashed working directory changes, but also restores the index.

If you asked Git to record a state of the path in the staging area, but changed your mind, you can reset the staged contents of the file to the committed version with `git reset HEAD -- <file>`.

If you mis-edited a file, so that the working directory version is a mess, and you want to restore it to the version from the index, use `git checkout -- <file>`. If you staged some of this mess, and would like to reset to the last committed version, use `git checkout HEAD -- <file>` instead.

Note

Actually these commands *do not really undo* operations; they restore the previous state based on a backup that is the worktree, the index, or the committed version. For example, if you staged some changes, modified a file, then added modifications to the staging area, you can reset the index to the committed version, but not to the state after the first and before the second `git add`.

Resetting a file to the old version

Of course, you can use any revision with a per-file `reset` and per-file `checkout`. For example, to replace the current worktree version of the `src/rand.c` file with the one from the previous commit, you can use `git checkout HEAD^ -- src/rand.c` (or redirect the output of `git show HEAD^:src/rand.c` to a file). To put the version from the `next` branch into the staging area, run `git reset next -- src/rand.c`.

Note: `git add <file>`, `git reset <file>`, and `git checkout <file>` all enter interactive mode for a given file with the `--patch` option. This can

be used to hand-craft a staged or worktree version of a file by selecting which changes should be applied (or un-applied).

Note

You might need to put a double dash -- before the file name here, if for example, you have a file with the same name as a branch.

Cleaning the working area

Untracked files and directories may pile up in your working directory. They can be left overs from merges, or be temporary files, proof of concept work, or perhaps mistakenly put there. Whatever the case, often there really is no pattern to them, and you don't need to make Git ignore them (see the *Ignoring files* section of this chapter); you just want to remove them. You can use `git clean` for this.

Because untracked files do not have a backup in the repository, and you cannot undo their removal (unless the operating system or file system supports undo), it's advisable to first check which files would be removed with `--dry-run` / `-n`. Actual removal by default requires the `--force` / `-f` option.

```
$ git clean --dry-run
Would remove patch-1.diff
```

Git will clean all untracked files recursively, starting from the current directory. You can select which paths are affected by listing them as an argument; you can also exclude additional types of file with the `--exclude=<pattern>` option. You can also interactively select which untracked files to delete with the `--interactive` option.

```
$ git clean --interactive
Would remove the following items:
  src/rand.c~
  screenlog.0
*** Commands ***
  1: clean          2: filter by pattern    3: select by numbers
  4: ask each       5: quit                  6: help
What now>
```

The `clean` command also allows us to only remove ignored files, for example, to remove build products but keep manually tracked files with the `-x` option (though usually it is better to leave removing build byproducts to the build system, so that cleaning the project files works even without having to clone the repository).

You can also use `git clean -x` in conjunction with `git reset --hard`, to create a pristine working directory to test a clean build, by removing both ignored and not-ignored untracked files, and resetting tracked files to the committed version.

Tip

Dirty working directory

The working directory is considered clean if it is the same as the committed and staged version, and dirty if there are modifications.

Multiple working directories

Git for a long time allowed to specify where to find the administrative area of the repository (the `.git` directory) with the `git --git-dir= <path> <command>`, or the `GIT_DIR` environment variable, making it possible to work from the detached working directory.

To be able to reliably use multiple working directories sharing a single repository, we had to wait until version 2.5 of Git. With it, you can create a new linked work tree by using `git worktree add <path> <branch>`, allowing us to have more than one branch checked out. For convenience, if you omit the `<branch>` argument, then a new branch will be created based on the name of the created worktree.

Note

If you use an older Git version, there is always the `git-new-workdir` script, which can be found in the `contrib/` area of the Git project repository. It is however, Unix-only (it relies on symbolic links), and is somewhat fragile.

This mechanism can be used instead of `git stash` if you need to switch to a different branch (for example, to urgently fix a security bug), but your current working directory, and possibly also the staging area, is in a state of high disarray. Instead of disturbing it, you create a temporary linked working tree to make a fix, and remove it when done.

This is an evolving area—consult the Git documentation for more information.

Summary

In this chapter we have learned how to better manage the contents of the working directory, and the contents of the staging area, preparing to create a new commit.

We know how to undo the last commit, how to drop changes to the working area, how to retroactively change the branch we are working on, and other uses of the `git reset` command. We now understand the three (and a half) forms of the reset.

We have learned how to examine and search the contents of the working directory, the staging area, and committed changes. We know how to use Git to copy the file version from the worktree, the index, or the HEAD into the worktree or the index. We can use Git to clean (remove) untracked files.

This chapter showed how to configure the handling of files in the working directory; how to make Git ignore files (by making them intentionally untracked) and why. It described how to handle the differences between line ending formats between operating systems. It explained how to enable (and write) keyword expansion, how to configure the handling of binary files, and enhance viewing the diff and merging specific classes of files.

We learned to stash away changes to handle interruptions, and to make it possible to test interactively prepared commits, before creating a commit. This chapter explained how Git manages stashes, enabling us to go beyond built-in operations.

This chapter, together with [Chapter 3, *Developing with Git*](#), taught how to create your contribution to a project; together with [Chapter 2, *Exploring Project History*](#), it also taught how to examine your clone of a project's repository.

The following chapters will teach you how to collaborate with other

people, how to send what you contributed, and how to merge changes from other developers.

Chapter 5. Collaborative Development with Git

Previous chapters, [Chapter 3, Developing with Git](#), and [Chapter 4, Managing Your Worktree](#), taught you how to make a new contributions to a project, but limited it to affecting only your own clone of the project's repository. The former chapter described how to commit new revisions, while the latter showed how Git can help you prepare it.

This chapter will present a *bird's-eye view* of various ways to collaborate, showing centralized and distributed workflows. It will focus on the repository-level interactions in collaborative development, while the set-up of branches will be covered in the next chapter, [Chapter 6, Advanced Branching Techniques](#).

This chapter will describe different collaborative workflows, explaining the advantages and disadvantages of each. You will also learn here the *chain of trust* concept, and how to use signed tags, signed merges, and signed commits.

The following topics will be covered in this chapter:

- Centralized and distributed workflows, and bare repositories
- Managing remotes and one-off single-shot collaboration
- Push, pull requests, and exchanging patches
- Using bundles for off-line transfer (sneakernet)
- How versions are addressed—the chain of trust
- Tagging, lightweight tags versus signed tags
- Signed tags, signed merges, and signed commits

Collaborative workflows

There are various *levels of engagement* when using a version control system. One might only be interested in using it for archaeology. [Chapter 2, Exploring Project History](#), will help with this. Of course, examining

project's history is an important part of development, too.

One might use version control for your private development, for a single developer project, on a single machine. [Chapter 3](#), *Developing with Git*, and [Chapter 4](#), *Managing Your Worktree*, show how to do this with Git. Of course, your own development is usually part of a collaboration.

But one of the main goals of version control systems is to help multiple developers work together on a project, collaboratively. Version control makes it possible to work simultaneously on a given piece of software in an effective way, ensuring that their changes do not conflict with each other, and helps with merging those changes together.

One might work on a project together with a few other developers, or with many. One might be a contributor, or a project maintainer; perhaps the project is so large that it needs subsystem maintainers. One might work in tight software teams, or might want to make it easy for external contributors to provide proposed changes (for example, to fix bugs, or an error in the documentation). There are various different workflows that are best suited for those situations:

- Centralized workflow
- Peer-to-peer workflow
- Maintainer workflow
- Hierarchical workflow

Bare repositories

There are two types of repositories: an ordinary non-bare one, with a working directory and a staging area, and a **bare repository**, bereft of the working directory. The former type is meant for private solo development, for creating new history, while the latter type is intended for collaboration and synchronizing development results.

By convention, bare repositories use the `.git` extension—for example, `project.git`—while non-bare repositories don't have it—for example, `project` (with the administrative area and the local repository in `project/.git`). You can usually omit this extension when cloning,

pushing to, or fetching from the repository; using either
`http://git.example.com/project.git` or
`http://git.example.com/project` as the repository URL will work.

To create the bare repository, you need to add the `--bare` option to the `init` or the `clone` command:

```
$ git init --bare project.git
Initialized empty Git repository in /home/user/project.git/
```

Interacting with other repositories

After creating a set of revisions, an extension to the project's history, you usually need to share it with other developers. You need to synchronize with other repository instances, publish your changes, and get changes from others.

From the perspective of the local repository instance, of your own clone of repository, you need to *push* your changes to other repositories (either the repository you cloned from, or your public repository), and *fetch* changes from other repositories (usually the repository you cloned from). After fetching changes, you sometimes need to incorporate them into your work, *merging* two lines of development (or *rebasing*)—you can do it in one operation with *pull*.

Usually you don't want your local repository to be visible to the public, as such repository is intended for private work (keeping work not ready yet from being visible). This means that there is an additional step required to make your finished work available; you need to publish your changes, for example with `git push`. The following diagram demonstrates creating and publishing commits, an extension of the one in [Chapter 3, Developing with Git](#). The *arrows* show Git commands to copy contents from one place to another, including to and from the remote repository.

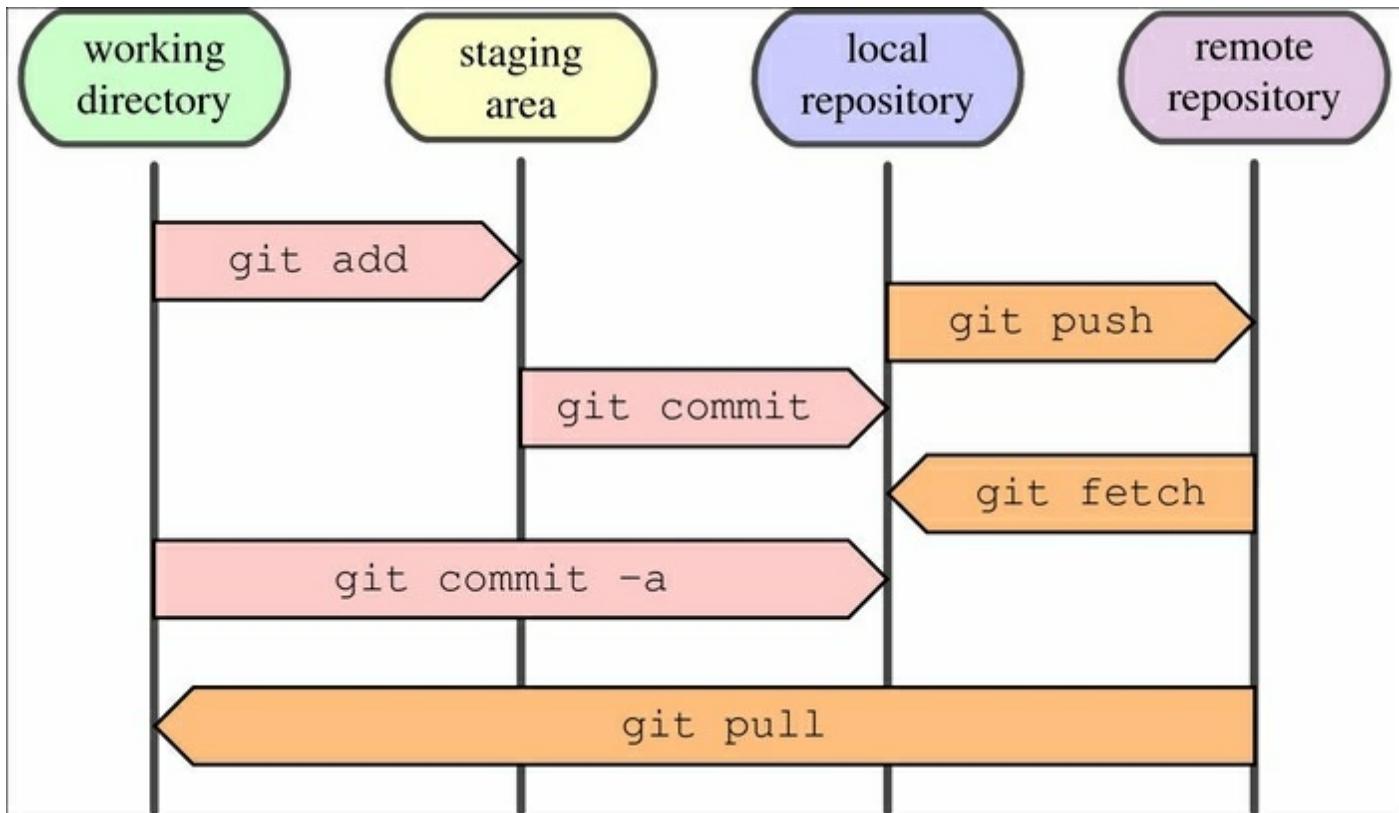


Fig 1: Creating and publishing commits.

The centralized workflow

With distributed version control systems you can use different collaboration models, more distributed or less distributed. In a centralized workflow, there is one central hub, usually a bare repository, that everyone uses to synchronize their work:

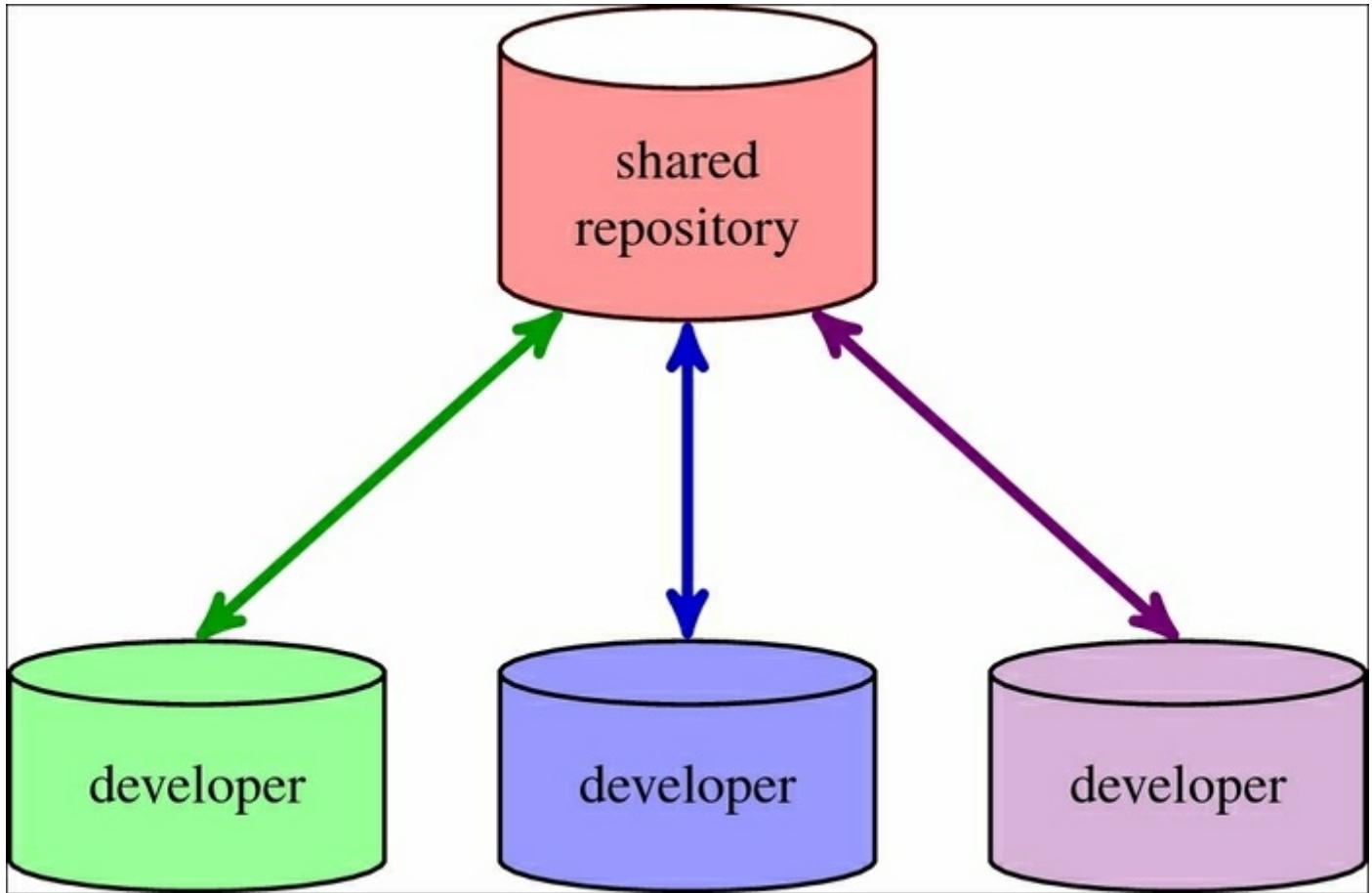


Fig 2: Centralized workflow. The shared repository is bare. The color of the line represents from which repository the transport is initiated; for example, a green line means that the command was invoked from within green repository, by its developer.

Each developer has his or her own non-bare clone of the central repository, which is used to develop new revisions of software. When changes are ready, they push those changes to the central repository, and fetch (or pull) changes from other developers from the central shared repository, so integration is distributed. This workflow is shown in Fig 2. The advantages and disadvantages of a centralized workflow are as follows:

- The advantage is its simple setup; it is a familiar paradigm for people coming from centralized version control systems and centralized management, and provides centralized access control and backup. It might be a good setup for a private project with a small team.

- The disadvantages are that the shared repository is a single point of failure (if there are problems with the central repository, then there is no way to synchronize changes), and that each developer pushing changes (making them available for other developers) might require updating one's own repository first and merging changes from others. You need also to trust developers with access to the shared repository in this setup.

The peer-to-peer or forking workflow

The opposite of a centralized workflow is a **peer-to-peer** or **forking workflow**. Instead of using a single shared repository, each developer has a public repository (which is bare), in addition to a private working repository (with a working directory), like in the following figure:

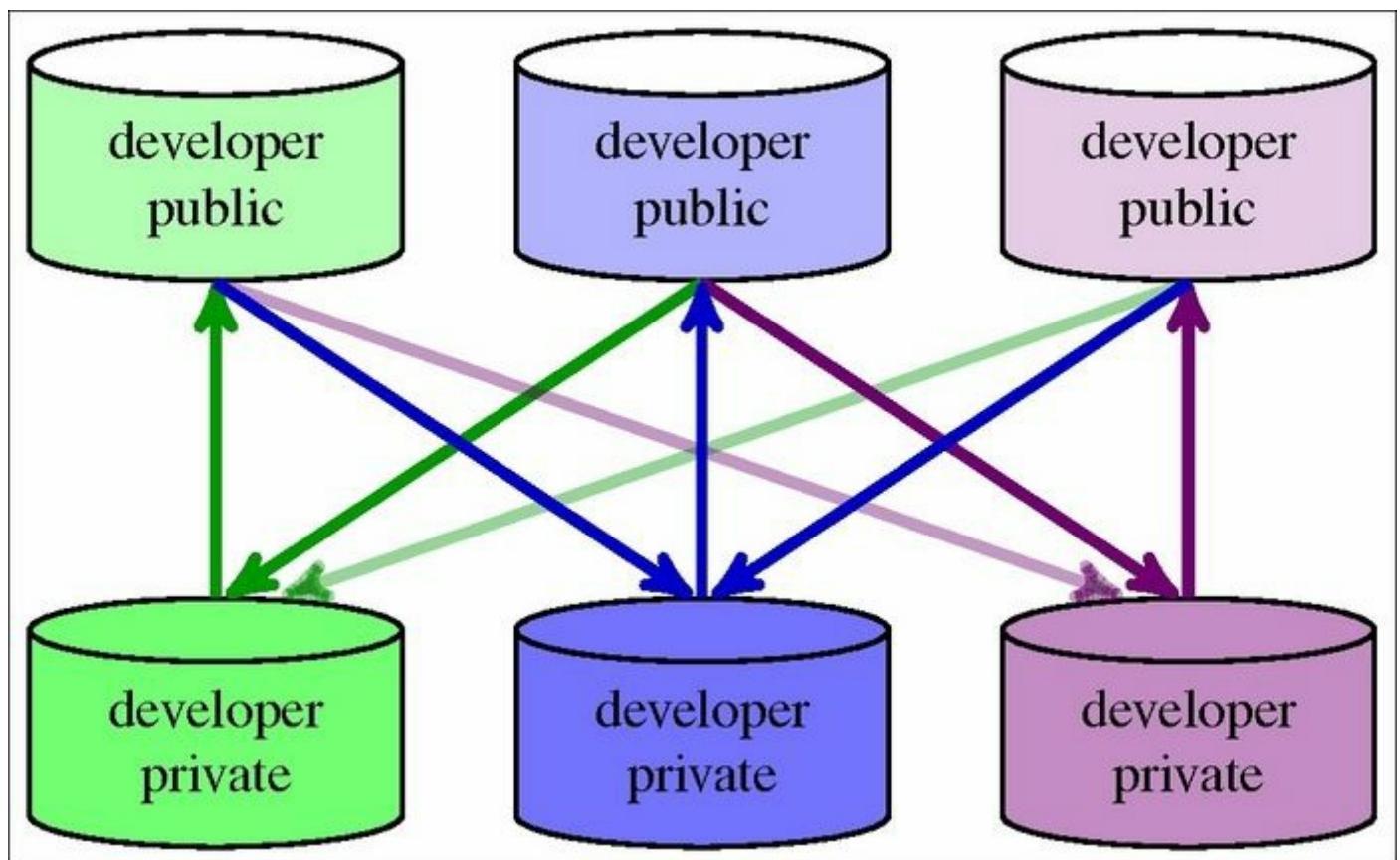


Fig 3: Peer-to-peer (forking) workflow. Each developer has his/her own private non-bare and their own public bare repository. The line color represents who did the transfer (who ran the command). Lines pointing up are push, lines pointing down are fetch.

When changes are ready, developers push to their own public repositories. To incorporate changes from other developers, one needs to fetch them from the public repositories of other developers. The advantages and disadvantages of the peer-to-peer or forking workflow are as follows:

- One advantage of the forking workflow is that contributions can be integrated without the need for a central repository; it is a fully distributed workflow. Another advantage is that you are not forced to integrate if you want to publish your changes; you can merge at your leisure. It is a good workflow for organic teams without requiring much setup.
- The disadvantages are a lack of the *canonical* version, no centralized management, and the fact that in this workflow base form you need to interact with many repositories (though `git remote update` can help here, doing multiple fetches with a single command.). Setup requires that developer public repositories need to be *reachable* from other developers' workstations; this might not be as easy as using one's own machine as a server for one's own public repositories. Also, as can be seen in *Fig 3*, collaboration gets more complicated with the growing number of developers.

The maintainer or integration manager workflow

One of the problems with peer-to-peer workflow was that there was no canonical version of a project, something that non-developers can use. Another was that each developer had to do his or her own integration. If we promote one of the public repositories in *Fig 3* to be canonical (official), and make one of the developers responsible for integration, we arrive at the **integration manager workflow (or maintainer workflow)**. The following diagram shows this workflow, with bare repositories at the top and non-bare at the bottom:

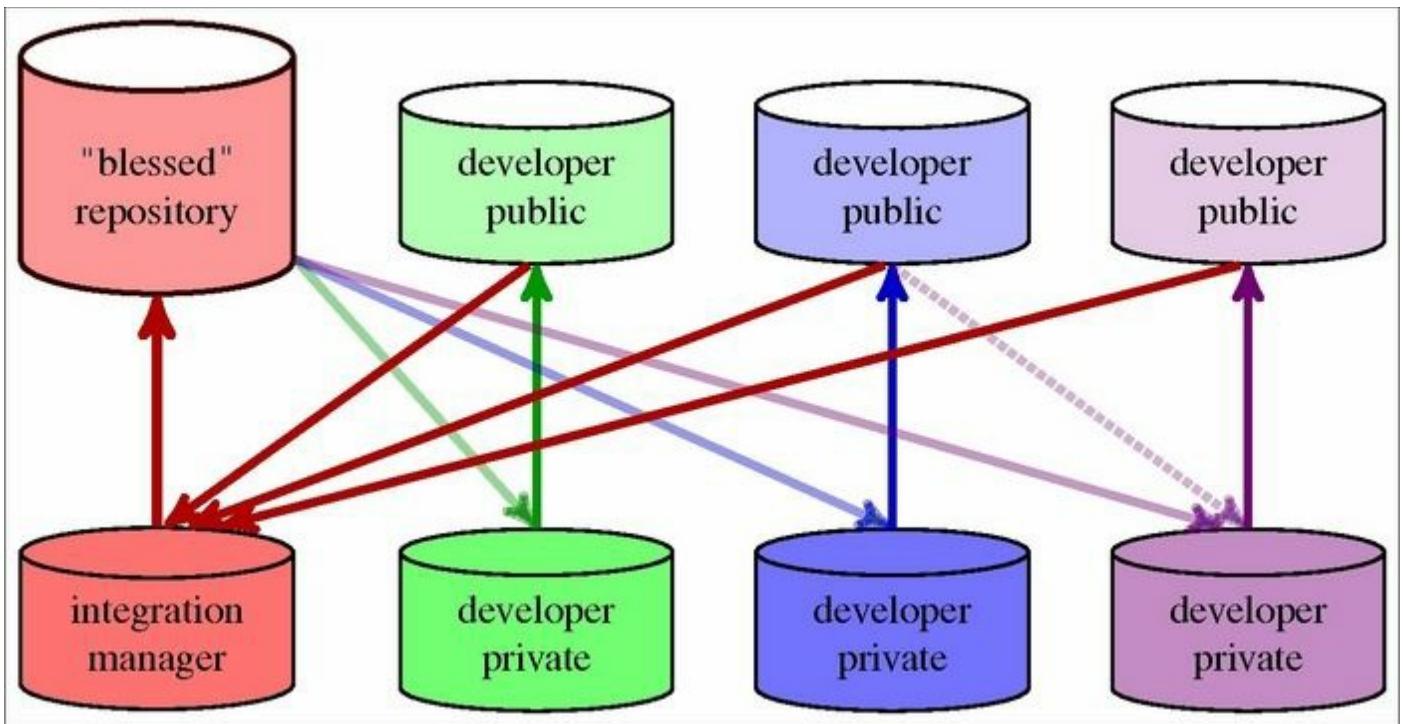


Fig 4: Integration-manager (maintainer) workflow. One of the developers has the role of integration manager, and his or her public repository is "blessed" as the official repository for a project. Incoming lines of the same color denote fetching; outgoing lines denote push. Dotted lines show the possibility of fetching from a non-official repository (for example, collaboration within a smaller group of developers).

In this workflow, when changes are ready, the developer pushes them to his or her own public repository, and tells the maintainer (for example via a pull request) that they are ready. The maintainer pulls changes from the developer's repository into own working repository and integrates the changes. Then the maintainer pushes merged changes to the blessed repository, for all to see. The advantages and disadvantages are as follows:

- The advantages are having an official version of a project, and that developers can continue to work without doing or waiting for integration, as maintainers can pull their changes at any time. It is a good workflow for a large organic team, like in open source projects. The fact that the blessed repository is decided by social

consensus allows an easy switch to other maintainers, either temporarily (for example, time off) or permanently (forking a project).

- The disadvantage is that for large teams and large projects the ability of the maintainer to integrate changes is a bottleneck. Thus, for very large organic teams, such as in Linux kernel development, it is better to use a hierarchical workflow.

The hierarchical or dictator and lieutenants workflows

The **hierarchical workflow** is a variant of the blessed repository workflow, generally used by huge projects with hundreds of collaborators. In this workflow, the project maintainer (sometimes called the **benevolent dictator**) is accompanied by additional integration managers, usually in charge of certain parts of the repository (subsystems); they're called **lieutenants**. The benevolent dictator's public repository serves as the blessed reference repository from which all the collaborators need to pull. Lieutenants pull from developers, the maintainer pulls from lieutenants, as shown in the following figure:

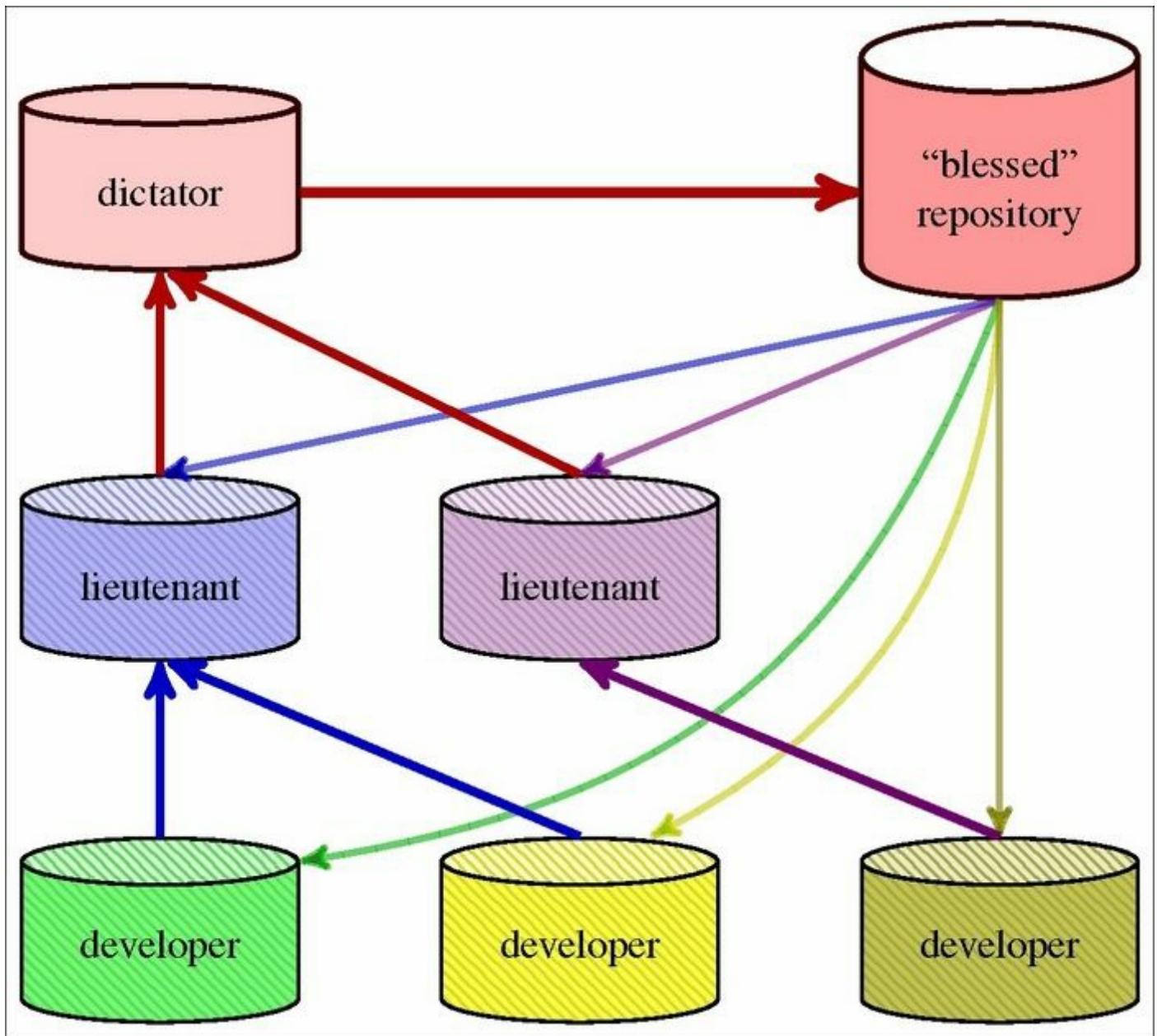


Fig 5. Dictator and lieutenants (hierarchical) workflow. There is an overall maintainer for the whole project, called dictator (whose public repository is official, "blessed" repository of a project), and subsystem integration managers, called lieutenants. Dashed pattern repositories are actually a pair of private and public repositories of a developer or a lieutenant. The person that initiates transfer is shown via line color.

In dictator and lieutenant workflows, there is a hierarchy (a network) of repositories. Before starting work: either development or merging, one would usually pull updates from the canonical (blessed) repository for a project. Developers prepare changes in their own private repository,

then send changes to an appropriate subsystem maintainer (lieutenant). Changes can be sent as patches in email, or by pushing them to the developer's public repository and sending a pull request.

Lieutenants are responsible for merging changes in their respective area of responsibility. The master maintainer (dictator) pulls from lieutenants (and occasionally directly from developers). The dictator is also responsible for pushing merged changes to the reference (canonical) repository, and usually also for release management (for example, creating tags for releases). The advantages and disadvantages are as follows:

- The advantage of this workflow is that it allows the project leader (the dictator) to delegate much of the integration work. This can be useful in very big projects (with respect to the number of developers and/or changes), or in highly hierarchical environments. Such workflow is used to develop Linux kernel.
- Its complicated setup is a disadvantage of this workflow. It is usually overkill for an ordinary project.

Managing remote repositories

When collaborating on any project managed with Git, you will interact often with a constant set of other repositories; for example, in an integration-manager workflow it would be (at least) the canonical blessed repository of a project. In many cases, you will interact with more than one remote repository.

Git allows us to save the information about a remote repository (in short: **remote**) in the `config` file, giving it a nickname (a shorthand name). Such information can be managed with the `git remote` command.

Note

There are also two legacy mechanisms to store repository shorthand:

- A named file in `.git/remotes`—the name of this file will be shorthand for remote. This file can contain information about the URL or URLs, and fetch and push refsspecs.
- A named file in `.git/branches`—the name of this file will be shorthand for remote. The contents of this file are just an URL for the repository, optionally followed by # and the branch name.

Neither of those mechanisms is likely to be found in modern repositories. See the *Remotes* section in the `git-fetch(1)` manpage for more details.

The origin remote

When cloning a repository, Git will create one remote for you—**the origin remote**, storing information about where you cloned from—that is the origin of your copy of the repository (hence the name). You can use this remote to fetch updates.

This is the default remote; for example `git fetch` without the remote name will use the origin remote; unless it is specified otherwise by the `remote.default` configuration variable, or unless the configuration for

the current branch (`branch.<branchname>.remote`) specifies otherwise.

Listing and examining remotes

To see which remote repositories you have configured, you can run the `git remote` command. It lists the shortnames of each remote you've got. In a cloned repository you will have at least one remote: `origin`.

```
$ git remote  
origin
```

To see the URL together with remotes, you can use `-v / --verbose` option:

```
$ git remote --verbose  
origin  git://git.kernel.org/pub/scm/git/git.git (fetch)  
origin  git://git.kernel.org/pub/scm/git/git.git (push)
```

If you want to inspect remotes to see more information about a particular remote, you can use the `git remote show <remote>` subcommand:

```
$ git remote show origin  
* remote origin  
  Fetch URL: git://git.kernel.org/pub/scm/git/git.git  
  Push URL: git://git.kernel.org/pub/scm/git/git.git  
  HEAD branch: master  
  Remote branches:  
    maint tracked  
    master tracked  
    next tracked  
    pu tracked  
    todo 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)
```

Git will consult the remote configuration, the branch configuration, and the remote repository (for an up-to-date status). If you want to skip contacting the remote repository and use cached information instead, add the `-n` option to `git remote show`.

As the information about remotes is stored in the repository configuration file, you can simply examine `.git/config`:

```
[remote "origin"]
  fetch = +refs/heads/*:refs/remotes/origin/*
  url = git://git.kernel.org/pub/scm/git/git.git
```

The difference between local and remote branches (and remote tracking branches: local representations of remote branches) will be described in [Chapter 6, Advanced Branching Techniques](#), together with an explanation of **refspecs**, as in `+refs/heads/*:refs/remotes/origin/*` in the preceding example.

Adding a new remote

To add a new remote Git repository and to store its information under a shortname, run `git remote add <shortname> <URL>`:

```
$ git remote add alice https://git.company.com/alice/random.git
```

Adding remote doesn't fetch from it automatically—you need to use the `-f` option for that (or run `git fetch <shortname>`).

This command has a few options that affect how Git creates a new remote. You can select which branches in the remote repository you are interested in with the `-t <branch>` option. You can change which branch is the default one in the remote repository (and which you can refer to by the remote name), using the `-m <branch>` option. Or you can configure the remote repository for mirroring rather than for collaboration with `--mirror=push` or `--mirror=fetch`.

For example, running the command:

```
$ git remote add -t master -t next -t maint github \
https://github.com/jnareb/git.git
```

will result in the following configuration of the remote:

```
[remote "github"]
  url = https://github.com/jnareb/git.git
```

```
fetch = +refs/heads/master:refs/remotes/github/master
fetch = +refs/heads/next:refs/remotes/github/next
fetch = +refs/heads/maint:refs/remotes/github/maint
```

Updating information about remotes

The information about the remote repository is stored in three places: in the remote configuration: `remote.<remote name>`, in remote-tracking branches, in remote-HEAD (`refs/remotes/<remote name>/HEAD` is a symref that denotes the default remote-tracking branch; that is, the remote tracking branch that `<remote name>` used as a branch expands to), and optionally the per-branch configuration: `branch.<branch name>`.

You could manipulate this information directly—either by editing the appropriate files or using manipulation commands such as `git config` and `git symbolic-ref`—but Git provides various `git remote` subcommands for this.

Renaming remotes

Renaming remote—that is, changing its nickname—is quite a complicated operation. Running `git remote rename <old> <new>` would not only change the section name in `remote.<old>`, but also the remote-tracking branches and accompanying refspec, their reflogs (if there are any—see the `core.logAllRefUpdates` configuration variable), and the respective branch configuration.

Changing the remote URLs

You can add or replace the URL for a remote with `git remote set-url`, but it is also quite easy to simply directly edit the configuration.

You can also use the `insteadOf` (and `pushInsteadOf`) configuration variables. This can be useful if you want to temporarily use another server, for example if the canonical repository is temporarily down. Say that you want to fetch Git from the repository on GitHub, because <https://www.kernel.org/> is down; you can do this by adding the following text to the `config` file:

```
[url "https://github.com/git/git.git"]
  insteadOf = git://git.kernel.org/pub/scm/git/git.git
```

Another use case for this feature is handling repository migration; you can use `insteadOf` rewriting in the per-user configuration file `~/.gitconfig` (or `~/.config/git/config`) without having to change the URL in each and every per-repository `.git/config` file. In the case of more than one match, the longest match is used.

Tip

Multiple URLs for a remote

You can set multiple URLs for a remote. Git will try all those URLs sequentially when fetching, and use the first one that works; when pushing, Git will publish to all URLs (all servers) simultaneously.

Changing the list of branches tracked by remote

A similar situation to changing the URL is with changing the list of branches tracked by a remote (that is, the contents of `fetch` lines): you can use `git remote set-branches` (with a sufficiently modern Git), or edit the config file directly.

Note that freeing a branch in a remote repository from being tracked does not remove the remote tracking branch—the latter is simply no longer updated.

Setting the default branch of remote

Having a default branch on remote is not required, but it lets us use the remote name (for example, `origin`) to be specified in lieu of a specific remote-tracking branch (for example, `origin/master`). This information is stored in the symbolic ref `<remote name>/HEAD` (for example, `origin/HEAD`).

You can set it with `git remote set-head`; the `--auto` option does that based on what is the current branch in the remote repository:

```
$ git remote set-head origin master
```

```
$ git branch -r
origin/HEAD -> origin/master
origin/master
```

You can delete the default branch on the remote with the `--delete` option.

Deleting remote-tracking branches

When a public branch is deleted in the remote repository, Git nevertheless keeps the corresponding remote-tracking branch. It does that because you might want to do, or might have done, your own work on top of it. You can however delete the remote tracking branch with `git branch -r -d`, or you can ask Git to prune all stale remote tracking branches under the remote with `git remote prune`. Or you can configure Git to do it automatically on fetch, as if `git fetch` were run with the `--prune` option, by setting the `fetch.prune` and `remote.<name>.prune` configuration variables.

You can check which remote tracking branches are stale with the `--dry-run` option to `git remote prune`, or with the `git remote show` command.

Deleting remote as a whole is as simple as running `git remote delete` (or its alias `git remote rm`). It also removes remote-tracking branches for the deleted remote.

Support for triangular workflows

In many collaborative workflows, like for example the maintainer (or integration manager) workflow, you fetch from one URL (from the blessed repository) but push to another URL (to your own public repository). See *Fig 4*: the developer interacts with three repositories—he or she fetches from the blessed repository (light red) into the developer private repository (darker), then pushes his or her work into the developer public repository (lighter).

In such a **triangular workflow** (three repositories), the remote you

fetch or pull from is usually the default `origin` remote (or `remote.default`). One option for configuring which repository you push to is to add this repository as a separate remote, and perhaps also set it up as the default with `remote.pushDefault`.

```
[remote "origin"]
  url = https://git.company.com/project
  fetch = +refs/heads/*:refs/remotes/origin/*
[remote "myown"]
  url = git@work.company.com:user/project
  fetch = +refs/heads/*:refs/remotes/myown/*
[remote]
  pushdefault = myown
```

You could also set it as `pushremote` in the per-branch configuration:

```
[branch "master"]
  remote = origin
  pushremote = myown
  merge = refs/heads/master
```

Another option is to use a single remote (perhaps even `origin`), but set it up with a separate `pushurl`. This solution however has the slight disadvantage that you don't have separate remote-tracking branches for the push repository (and thus there is no support `@{push}` notation in addition to having `@{upstream}` as a shortcut for specifying the appropriate remote-tracking branches; however, the former has only been available since Git 2.5.0):

```
[remote "origin"]
  url = https://git.company.com/project
  pushurl = git@work.company.com:user/project
  fetch = +refs/heads/*:refs/remotes/origin/*
```

Transport protocols

In general, URLs in the configuration of remote contain information about the transport protocol, the address of the remote server (if any), and the path to the repository. Sometimes, the server that provides access to the remote repository supports various transport protocols; you need to select which one to use. This section is intended to help with this choice.

Local transport

If the other repository is on the same local filesystem, you can use the following syntaxes for specifying the URL:

```
/path/to/repo.git/  
file:///path/to/repo.git/
```

The former implies the `--local` option to the Git clone, which bypasses the smart Git-aware mechanism and simply makes a copy (or a hardlink for immutable files under `.git/objects`, though you can avoid this with the `--no-hardlinks` option); the latter is slower but can be used to get a clean copy of a repository.

This is a nice option for quickly grabbing work from someone else's working repository, or for sharing work using a shared filesystem with the appropriate permissions.

As a special case, a single dot `"."` denotes the current repository. This means that

```
$ git pull . next
```

is roughly equivalent to

```
$ git merge next
```

Tip

Legacy (dumb) transports

Some transports do not require any Git-aware smart server—they don't need Git installed on the server (for smart transports at least `git-upload-pack` and/or `git-receive-pack` is needed), though most of them do need extra information generated by `git update-server-info` alongside the repository refs, objects, and packfiles (copied in some way).

Rsync protocol transport – the unsafe one

One of the old protocols that Git supported from the very beginning, allowing us to fetch and push, read and write to the remote repository, is the `rsync` protocol, using the following URL type:

```
rsync://host.example.com/path/to/repo.git/
```

The `rsync` protocol is deprecated, because it does not ensure proper ordering when getting data; if you fetch from a non-quiescent repository, you can get *invalid data*. On the other hand it is quite fast and actually resumable. However, if you have a problem doing the initial clone on an unreliable network, it is better to use bundles rather than rsync protocol, as described next in the part about dumb HTTP.

FTP(S) and dumb HTTP(S) protocol transports – the ineffective ones

These transports need only the appropriate stock server (an FTP server or a web server), and up-to-date data from `git update-server-info`. When fetching from such a server, Git uses the so-called **commit walker** downloader: going down from fetched branches and tags, Git walks down the commit chain, and downloads objects or packs containing missing revisions and other data (for example, file contents at revision).

This transport is inefficient (in terms of bandwidth, but especially in terms of latency), but on the other hand it can be resumed if interrupted. Nevertheless there are better solutions than using dumb protocols, namely involving bundles (see the *Offline transport with bundles*

section in this chapter), when the network connection to the server is unreliable enough that you can't get the clone.

Pushing to a dumb server is possible only via the HTTP and HTTPS protocols, requires the web server to support WebDAV, and Git has to be built with the expat library linked. The FTP and FTPS protocols are read-only (supporting only `clone`, `fetch`, and `pull`).

Smart transports

When the repository you want to fetch from is on another machine, you need to access the Git server. Nowadays most commonly encountered are Git-aware smart servers. The smart downloader negotiates which revisions are necessary, and creates a customized packfile to send to a client. Similarly, during the push the Git on the server talks to the Git on the user's machine (to the client) to find which revisions to upload.

Git-aware smart servers use the `git upload-pack` downloader for fetching and the `git receive-pack` for pushing. You can tell Git where to find them if they are not in `PATH` (but for example are installed in one's home directory) with the `--upload-pack` and `--receive-pack` options for fetch and push, or the `uploadpack` and `receivepack` remote configuration.

With very few exceptions (such as the repository using submodules accessed by an ancient Git that does not understand them), Git transport is backward- and forward-compatible—the client and server negotiate what capabilities they can both use.

Native Git protocol

The native transport, using `git://` URLs, provides read-only anonymous access (though you could in principle configure Git to allow pushing by enabling the `receive-pack` service, either from the command line or via the `daemon.receivePack` boolean-valued configuration variable; using this mechanism is not recommended at all, even in a closed local network).

Git protocol does no authentication, including no server authentication, and should be used with caution on unsecured networks. The `git` daemon TCP server for this protocol normally listens on port 9418; you need to be able to access this port (through the firewall) to be able to use the native Git protocol.

SSH protocol

The **Secure SHell (SSH)** transport protocol provides authenticated read-write access. Git simply runs `git upload-pack` or `git receive-pack` on the server, using SSH to execute the remote command. There is no possibility for anonymous, unauthenticated access, though you could as workaround set up a `guest` account for it (password-less or with an empty password).

Using public-private key authentication allows access without requiring you to provide a password on every connection, with the only possible exception of providing it once: to unlock a password-protected private key. You can read more about authentication in the *Credentials/password management* section.

For SSH protocol you can use the URL syntax with `ssh://` as the protocol part:

```
ssh://[user@]host.example.com[:port]/path/to/repo.git/
```

Alternatively you can use the `scp`-like syntax:

```
[user@]host.example.com:path/to/repo.git/
```

The SSH protocol additionally supports the `~username` expansion, just like the native Git transport (`~` is the home directory of the user you log in as, `~user` is the home directory of `user`), in both syntax forms:

```
ssh://[user@]host.example.com[:port]/~[user]/path/to/repo.git/  
[user@]host.example.com:~[user]/path/to/repo.git/
```

SSH uses the first contact authentication for servers—it remembers the key that the server side previously used, and warns the user if it has

changed, asking then for confirmation (the server key could have been changed legitimately, for example due to a SSH server reinstall). You can check the server key fingerprint on the first connection.

Smart HTTP(S) protocol

Git also supports the smart HTTP(S) protocol, which requires a Git-aware CGI or server module—for example, `git-http-backend` (itself a CGI module). As a design feature, Git can automatically upgrade dumb protocol URLs to smart URLs. Conversely, a Git-aware HTTP server can downgrade to the backward-compatible dumb protocol (at least for fetching: it doesn't support WebDAV-based dumb HTTP push). This feature allows to use the same HTTP(S) URL for both dumb and smart access:

```
http[s]://[user@]host.example.com[:port]/path/to/repo.git/
```

By default, without any other configuration, Git allows anonymous downloads (`git fetch`, `git pull`, `git clone`, and `git ls-remote`), but requires that the client is authenticated for upload (`git push`).

Standard HTTP authentication is used if authentication is required to access a repository, which is done by the HTTP server software. Using SSL/TLS with HTTPS ensures that if the password is sent (for example, if the server uses Basic HTTP authentication), it is sent encrypted, and that the server identity is verified (using server CA certificates).

Offline transport with bundles

Sometimes there is no direct connection between your machine and the server holding the Git repository that you want to fetch from. Or perhaps there is no server running, and you want to copy changes to another machine anyway. Maybe your network is down. Perhaps you're working somewhere on-site and don't have access to the local network for security reasons. Maybe your wireless/Ethernet card just broke.

Enter the `git bundle` command. This command will package up everything that would normally be transferred over the wire, putting

objects and references into a special binary archive file called **bundle** (like packfile, only with branches and so on). You need to specify which commits are to be packed—something that network protocols do automatically for you for online transport.

Note

When you are using one of the smart transports, a want/have negotiation phase takes place, where the client tells the server what it has in its repository and which advertised references on the server it wants, to find common revisions. This is then used by the server to create a packfile and send the client only what's necessary, minimizing the bandwidth use.

Next you move this archive by some means (for example, by so called sneakernet, which means saving bundle to a removable storage and physically moving the media) to your machine. You can then incorporate the bundle contents by using `git clone` or `git fetch` with the filename of bundle in place of the repository URL.

Tip

Proxies for Git transports

When direct access to the server is not possible, for example, from within a firewalled LAN, sometimes you can connect via a proxy.

For the native Git protocol (`git://`), you can use the `core.gitProxy` configuration variable, or the `GIT_PROXY_COMMAND` environment variable to specify a proxy command—for example, `ssh`. This can be set on a per-remote basis with this special syntax for the `core.gitProxy` value: `<command> for <remote>`.

You can use the `http.proxy` configuration variable or `curl` environment variables to specify the HTTP proxy server to use for the HTTP(S) protocol (`http(s)://`). This can be set on a per-remote basis with the `remote.<remote name>.proxy` configuration variable.

You can configure SSH (using its configuration files, for example, `~/.ssh/config`) to use tunneling (port forwarding) or a proxy command (for example, the `netcat/nc`, or netcat mode of `ssh`). It is a recommended solution for the SSH proxy; if neither tunneling nor proxy is possible, you can use the `ext::` transport helper, as shown later in this chapter.

Cloning and updating with bundle

Let's assume that you want to transfer the history of a project (say, limited to the `master` branch for simplicity) from `machineA` (for example, your `work computer`) to `machineB` (for example, an `onsite computer`). There is however no direct connection between those two machines.

First, we create a bundle that contains the whole history of the `master` branch (see [Chapter 2, Exploring Project History](#)), and tag this point of history to know what we bundled, for later:

```
user@machineA ~$ cd repo
user@machineA repo$ git bundle create ../repo.bundle master
user@machineA repo$ git tag -f lastbundle master
```

Here the bundle file was created outside the working directory. This is a matter of choice; storing it out of the repository means that you don't have to worry about accidentally adding it to your project history, or having to add a new ignore rule. The `*.bundle` file extension is also a matter of the naming convention used.

Note

For security reasons, to avoid information disclosure about the parts of history that was deleted but not purged (for example, an accidentally committed file with a password), Git only allows fetching from `git show-ref-compatible` references: branches, remote-tracking branches, and tags.

The same restrictions apply when creating a bundle. This means for example that (for implementation reasons) you cannot run `git bundle create master^1`. Though of course, because you control the server end,

as a workaround you can create a new branch for `master^`, (temporarily) rewind `master`, or check out the detached `HEAD` at `master^`.

Then you transfer the just created `repo.bundle` file to `machineB` (via email, on a USB pen drive, on CD-R, and so on.). Because this bundle consists of a self-contained, whole subset of the history, down to the first (parent-less) root commit, you can create a new repository by cloning from it, putting the bundle filename in place of the repository URL:

```
user@machineB ~$ git clone repo.bundle repo
Initialized empty Git repository in /home/user/repo/.git/
warning: remote HEAD refers to non-existent ref, unable to
checkout.
```

```
user@machineB ~$ cd repo
user@machineB repo$ git branch -a
  remotes/origin/master
```

Oops. We didn't bundle `HEAD`, so `git clone` didn't know which branch is current and therefore should be checked out.

```
user@machineB repo$ git bundle list-heads ../repo.bundle
5d2584867fe4e94ab7d211a206bc0bc3804d37a9 refs/heads/master
```

Note

Because `bundle` can be treated as a remote repository, we could simply use `git ls-remote ../repo.bundle` here instead of `git bundle list-heads ../repo.bundle`.

Therefore, with this bundle being as it were, we need to specify which branch to check out (this would not be necessary if we had bundled `HEAD` too):

```
user@machineB ~$ git clone repo.bundle --branch master repo
```

Let's fix the problem with the lack of checkout (assuming that you use a modern enough Git):

```
user@machineB repo$ git checkout master
Branch master set up to track remote branch master from origin.
Already on 'master'
```

Note

Here we used a special case of `git checkout <branch>`—because the master branch does not exist, but there is a remote-tracking branch with the same name for exactly one remote (origin/master here), Git will assume that we meant to create a local branch for the development that is to be published to the master branch in the origin repository. With an older Git, we would need to specify this explicitly:

```
user@machineB repo$ git checkout -b master --track  
origin/master
```

This will define a remote called `origin`, with the following configuration:

```
[remote "origin"]  
  url = /home/user/repo.bundle  
  fetch = +refs/heads/*:refs/remotes/origin/*  
[branch "master"]  
  remote = origin  
  merge = refs/heads/master
```

To update the repository on `machineB` cloned from the bundle, you can fetch or pull after replacing the original bundle stored at `/home/user/repo.bundle` with the one with incremental updates.

To create a bundle containing changes since the last transfer in our example, go to `machineA` and run the following command:

```
user@machineA repo$ git bundle create ../repo.bundle \  
lastbundle..master  
user@machineA repo$ git tag -f lastbundle master
```

This will bundle all changes since the `lastbundle` tag; this tag denotes what was copied with the previous bundle (see [Chapter 2, Exploring Project History](#), for an explanation of double-dot syntax). After creating a bundle, this will update the tag (using `-f` to replace it), like it was done the first time when creating a bundle, so that the next bundle can also be created incrementally from the now current point.

Then you need to copy the bundle to `machineB`, replacing the old one. At this point one can simply pull to update the repository:

```
user@machineB repo$ git pull
From /home/user/repo.bundle
  ba5807e..5d25848  master      -> origin/master
Updating ba5807e..5d25848
Fast-forward
```

Using bundle to update an existing repository

Sometimes you might have a repository cloned already, only for the network to fail. Or perhaps you moved outside the local area network (LAN), and now you have no access to the server. End result: you have an existing repository, but no direct connection to the upstream (to the repository we cloned from).

Now if you don't want to bundle up the whole repository, like in the *Cloning and updating with bundle* section, you need to find some way to specify the cut-off point (base) in such a way that it is included in the target repository (on your machine). You can specify the range of revisions to pack into the bundle using almost any technique from [Chapter 2, Exploring Project History](#). The only limitation is that the history must start at a branch or tag (anything that `git show-ref` accepts). You can of course check the range with the `git log` command.

Commonly used solutions for specifying the range of revisions to pack into bundle are as follows:

- Use the tag that is present in both repositories:

```
machineA repo$ git bundle create ../repo.bundle
v0.1..master
```

- Create a cut-off based on the time of commit creation:

```
machineA repo$ git bundle create ../repo.bundle --
since=1.week master
```

- Bundle just the last few revisions, limiting the revision range by the number of commits:

```
machineA repo$ git bundle create ../repo.bundle -5 master
```

Note

Better to pack too much, than too little. Otherwise you get something like this:

```
user@machineB repo$ git pull ../repo.bundle master
error: Repository lacks these prerequisite commits:
error: ca3cdd6bb3fcd0c162a690d5383bdb8e8144b0d2
```

You can check if the repository has the requisite commits to fetch from bundle with `git bundle verify`.

Then, after transporting it to `machineB`, you can use the bundle file just like a regular repository to do a one-off pull (putting bundle filename in place of URL or remote name):

```
user@machineB repo$ git pull ../repo.bundle master
From ../repo.bundle
 * branch           master      -> FETCH_HEAD
Updating ba5807e..5d25848
```

If you don't want to deal with the merge, you can fetch into the remote-tracking branch (the `<remote branch>:<remote-tracking branch>` notation used here, which is known as `refspec`, will be explained in [Chapter 6, Advanced Branching Techniques](#)):

```
user@machineB repo$ git fetch ../repo.bundle \
  refs/heads/master:refs/remotes/origin/master
From ../repo.bundle
  ba5807e..5d25848  master      -> origin/master
Updating ba5807e..5d25848
```

Alternatively, you can use `git remote add` to create a new shortcut, using the path to the bundle file in place of the repository URL. Then you can simply deal with bundles as described in the previous section.

Utilizing bundle to help with the initial clone

Smart transports provide much more effective transport than dumb ones. On the other hand, the concept of a resumable clone using smart

transport remains elusive to this day (it is not available in Git version 2.7.0, though perhaps somebody will implement it in the future). For large projects with a long history and with a large number of files, the initial clone might be quite large (for example, `linux-next` is more than 800 MB) and take pretty long time. This might be a problem if the network is unreliable.

You can create a bundle from the source repository, for example with the following command:

```
user@server ~$ git --git-dir=/dir/repo.git bundle create --all  
HEAD
```

Some servers may offer such bundles to help with the initial clone. There is an emerging practice (a convention) that the repository with given URL has a bundle available at the same URL but with the `.bundle` suffix. For example, `https://git.example.com/git/repo.git` can have its bundle available at `https://git.example.com/git/repo.bundle`.

You can then download such a bundle, which is an ordinary static file, using any resumable transport: HTTP(S), FTP(S), rsync, or even BitTorrent (with the appropriate client that supports resuming the download).

Remote transport helpers

When Git doesn't know how to handle a certain transport protocol (which doesn't have built-in support), it attempts to use the appropriate remote helper for a protocol, if one exists. That's why an error within the protocol part of the repository URL looks like it does:

```
$ git clone shh://git@example.com:repo  
Cloning into 'repo'...  
fatal: Unable to find remote helper for 'shh'
```

This error message means that Git tried to find `git-remote-shh` to handle the `shh` protocol (actually a typo for `ssh`), but didn't find an executable with such a name.

You can explicitly request a specific remote helper with the `<transport>::<address>` syntax, where `<transport>` defines the helper (`git remote-<transport>`), and `<address>` is a string that the helper uses to find the repository.

Modern Git implements support for the dumb HTTP, HTTPS, FTP, and FTPS protocols with a `curl` family of remote helpers: `git-remote-http`, `git-remote-https`, `git-remote-ftp`, and `git-remote-ftps`, respectively.

Transport relay with remote helpers

Git includes two generic remote helpers that can be used to proxy smart transports: the `git-remote-fd` helper to connect to remote server via either a bidirectional socket or a pair of pipes, and the `git-remote-ext` helper to use an external command to connect to the remote server.

In the case of the latter, which uses the `"ext::<command> [<arguments>...]"` syntax for the repository URL, Git runs the specified command to connect to the server, passing data for the server to the standard input of the command, and receiving a response on its standard output. This data is assumed to be passed to a `git://` server, `git-upload-pack`, `git-receive-pack`, or `git-upload-archive` (depending on the situation).

For example, let's assume that you have your repository on a LAN host where you can log in using SSH. However, for security reasons this host is not visible on the Internet, and you need to go through the gateway host: `login.example.com`.

```
user@home ~$ ssh user@login.example.com
user@login ~$ ssh work
user@work ~$ find . -name .git -type d -print
./repo/.git
```

The trouble is that, also for security reasons, this gateway host either doesn't have Git installed (reducing the attack surface), or doesn't have your repository present (it uses a different filesystem). This means that you cannot use the ordinary SSH protocol. But the SSH transport is just `git-receive-pack` / `git-upload-pack` accessed remotely via SSH, with

the path to the repository as a parameter. This means that you can use the `ext::` remote helper:

```
user@home ~$ git clone \
    "ext::ssh -t user@login.example.com ssh work %S 'repo'" repo
Cloning into 'repo'...
Checking connectivity... done.
```

Here, `%S` will be expanded by Git into the full name of the appropriate service—`git-upload-pack` for fetching and `git-receive-pack` for the push. The `-t` option is needed if logging to the internal host uses interactive authentication (for example, a password). Note that you need to give the name (`repo`, here) to the result of cloning; otherwise, Git will use the command (`ssh`) as the repository name.

Note

You can also use `"ext::ssh [<parameters>...] %S '<repository>'"` to use specific options for SSH transport—for example, selecting the keypair to use - without needing to edit `.ssh/config`.

This is not the only possible solution—though there is no built-in support for sending the SSH transport through a proxy, like there is for native `git://` protocol (among others, `core.gitProxy`) and for HTTP (among others, `http.proxy`), you can however do it via configuring the SSH example in `.ssh/config` (see `ProxyCommand`), or by creating a SSH tunnel.

On the other hand, you can use the `ext::` remote helper also to proxy the `git://` protocol—for example, with the help of `socat`—including using a single proxy for multiple servers. See the `git-remote-ext(1)` manpage for details and examples.

Using foreign SCM repositories as remotes

The remote helper mechanism is very powerful. It can be used to interact with other version control systems, transparently using their repositories as if they were native Git repositories. Though there is no such built-in helper (unless you count the `contrib/` area in the Git

sources), you can find `git-remote-hg` (or `gitifyhg`) helper to access Mercurial repositories, and `git-remote-bzr` to access Bazaar repositories.

Once installed, those remote helper bridges will allow you to `clone`, `fetch`, and `push` to and from the Mercurial or Bazaar repositories as if they were Git ones, using the `<helper>:<URL>` syntax. For example, to clone Mercurial repository you can simply run the following command:

```
$ git clone "hg::http://hg.example.com/repo"
```

There is also the `remote.<remote name>.vcs` configuration variable, if you don't like using the `<helper>::` prefix in the repository URL. With this method you can use the same URL for Git like for the original VCS (version control system).

Of course one needs to remember about impedance mismatches between different version control systems, and the limitations of the remote helper mechanism. There are some features that do not translate at all, or do not translate well—for example, *octopus* merges (with more than two parent commits) in Git, or multiple anonymous branches (heads) in Mercurial. With remote helpers there is also no place to fix mistakes, replace references to other revisions with target native syntax, and otherwise clean up artifacts created by repository conversions—as can and should be done with a one-time conversion when changing version control systems. (Such a clean-up can be done with, for example, the help of the `reposurgeon` third-party tool).

With remote helpers, you can even use things that are not version control repositories in the strict sense; for example, with the Git-Mediawiki project you can use Git to view and edit a MediaWiki-based wiki (for example, Wikipedia), treating the history of pages as a Git repository:

```
$ git clone mediawiki::http://wiki.example.com
```

Beside that, there are remote helpers that allow additional transport protocols, or storage options—such as the `git-remote-s3bundle` to store

the repository as a bundle file on Amazon S3.

Credentials/password management

In most cases, with the exception of the local protocol, publishing changes to the remote repository requires authentication (the user identifies itself) and authorization (the given user has permission to push) provided by Git. Sometimes, fetching the repository also requires authentication.

Commonly used **credentials** for authentication are username and password. You can put the username in the HTTP and SSH repository URLs, if you are not concerned about information leakage (in respect of valid usernames), or you can use the credential helper mechanism. You should *never* put passwords in URLs, even though it is technically possible for HTTP ones — the password can be visible to other people, for example when they are listing processes.

Besides the mechanism inherent in the underlying transport engine, be it `SSHASKPASS` for `ssh`, or the `~/.netrc` file for `curl`-based transport, Git provides its own integrated solution.

Asking for passwords

Some Git commands that interactively ask for a password (and a username if it is not known)—such as `git svn`, the HTTP interface, or IMAP authentication—can be told to use an external program. The program is invoked with a suitable prompt (a so-called domain, describing what the password is for), and Git reads the password from the standard output of this program.

Git will try the following places to ask the user for usernames and passwords; see the `gitcredentials(7)` manpage:

- The program specified by environment variable `GITASKPASS`, if set (Git-specific environment variables always have higher precedence than configuration variables)
- Otherwise, the `core.askpass` configuration variable is used, if set

- Otherwise, the `SSHASKPASS` environment variable is used, if set (not Git-specific)
- Otherwise, user is prompted on the terminal

This "askpass" external program is usually selected according to the desktop environment of the user (after installing it, if necessary). For example (`x11-ssh-askpass` provides a plain X-Window dialog asking for the username and password; there is `ssh-askpass-gnome` for GNOME, `ksshaskpass` for KDE, `mac-ssh-askpass` can be used for MacOS X, and `win-ssh-askpass` can be used for MS Windows. Git comes with a cross-platform password dialog in Tcl/Tk—`git-gui--askpass`—to accompany the `git gui` graphical interface and the `gitk` history viewer.

Tip

Git configuration precedence

Commands in Git have many ways to configure their behavior. They are applied in this order: the first existing specification wins, from the most specific to the least specific:

- Command line option, example, `--pager`,
- Git-specific environment variable, for example, `GIT_PAGER`, or `GITASKPASS` (such variables usually use the `GIT_` prefix)
- Configuration option (in one of the config files, with its own precedence), for example, `core.pager` or `core.askpass`
- A generic environment variable, for example, `PAGER` or `SSHASKPASS`
- The built-in default, for example, the `less` pager or terminal prompt.

Public key authentication for SSH

For the SSH transport protocol there are additional authentication mechanisms besides passwords. One of them is **public key authentication**. It is very useful to avoid being asked for a password over and over. Also, the repository hosting service providing the SSH access may require using it, possibly because identifying a user based on his or her public key doesn't require an individual account (that's what,

for example, gitolite uses).

The idea is that the user creates a public/private key pair by running, for example, ssh-keygen. The public key is then sent to the server, for example, using ssh-copy-id (which adds the public key *.pub at the end of the ~/.ssh/authorized_keys file on the remote server). You can then log in with your private key that is on your local machine, for example, as ~/.ssh/id_dsa. You might need to configure ssh (in ~/.ssh/config on Linux) to use a specific identity file for a given connection (hostname), if it is not the default identity key.

Another convenient way to use public key authentication is with an authentication agent such as ssh-agent (or Pageant from PuTTY on MS Windows). Utilizing an agent also makes it more convenient to work with passphrase-protected private keys—you need to provide the password only once, to the agent, at the time of adding the key (which might require user action, for example running ssh-add for ssh-agent).

Credential helpers

It can be cumbersome to input the same credentials over and over. For SSH, you can use public key authentication; there is no true equivalent for other transports. Git credential configuration provides two methods to at least reduce the number of questions.

The first is the static configuration of default usernames (if one is not provided in the URL) for a given **authentication context**, for example hostname:

```
[credential "https://git.example.com"]
  username = user
```

It helps if you don't have secure storage for credentials.

The second is to use external programs from which Git can request both usernames and passwords—**credential helpers**. These programs usually interface with secure storage (a keychain, keyring, wallet, credentials manager, and so on) provided by the desktop environment or the

operating system.

Git by default includes at least the `cache` and `store` helpers. The `cache` helper (`git-credential-cache`) stores credentials in memory for a short period of time; by default it caches usernames and passwords for 15 minutes. The `store` helper (`git-credential-store`) stores *unencrypted* credentials for indefinitely long time on disk, in files readable only by the user (similar to `~/.netrc`); there is also a third-party `netrc` helper (`git-credential-netrc`) for GPG-encrypted `netrc/authinfo` files.

Selecting a credential helper to use and its options, can be configured either globally or per-authentication context, as in the previous example. Global credentials configuration looks like this:

```
[credential]
    helper = cache --timeout=300
```

This will create Git cache credentials for 300 seconds (five minutes). If the credential helper name is not an absolute path (for example, `/usr/local/bin/git-kde-credentials-helper`), Git will prepend the `git credential-` prefix to the helper name. You can check what types of helper are available with `git help -a | grep credential-` (excluding those with a double dash `--` in the name—those are internal implementations).

There exist credential helpers that are using secure storage of the desktop environment. When you are using them, you need to provide the password only once, to unlock the storage (some helpers can be found in the `contrib/` area in Git sources). There is `git-credential-gnome-keyring` and `git-credential-gnomekeyring` for the Gnome Keyring, `git-credential-osxkeychain` for the MacOS X Keychain, and `git-credential-wincred` and `git-credential-winstore` for MS Windows' Credential Manager/Store.

Git will use credential configuration for the most specific authentication context, though if you want distinguish the HTTP URL by pathname (for example, providing different usernames to different repositories on the same host) you need to set the `useHttpPath` configuration variable to

true. If there are multiple helpers configured for context, each will be tried in turn, until Git acquires both a username and a password.

Note

Before the introduction of credential helpers, one could use `askpass` programs that interface with the desktop environment keychain, for example, `kwallettaskpass` (for KDE Wallet) or `git-password` (for the MacOS X Keychain).

Publishing your changes upstream

Now that the *Collaborative workflows* section has explained various repository setups, we'll learn about a few common patterns for contributing to a project. We'll see what our (main) options for publishing changes are.

Before starting work on new changes, you should usually sync to the main development, merging the official version into your repository. This, and the work of the maintainer, is left to be described in [Chapter 7, Merging Changes Together](#).

Pushing to a public repository

In a **centralized workflow**, publishing your changes consists simply of **pushing** them to the central server, as shown in *Fig 2*. Because you share this central repository with other developers, it can happen that somebody has already pushed to the branch you are trying to update (the non-fast-forward case). In this scenario, you need to pull (fetch and merge, or fetch and rebase) others' changes, before being able to push yours.

Another possible system with similar workflow is when your team submits each set of changes to the code review system, for example, Gerrit. One available option is to push to a special `ref` (which is named after a target branch, for example to `refs/for/<branchname>`) in a special repository. Then change review server makes each set of changes land automatically on a separate per-set ref (for example, `refs/changes/<change-id>` for commits belonging to a series with given Change-ID).

Tip

In both peer-to-peer (see *Fig 3*), and in maintainer workflows or its

hierarchical workflow variant (*Fig 4* and *Fig 5*), the first step in getting your changes included in the project is also to push, but to push to your own public repository. Then you need to ask your co-developers, or the project maintainer, to merge in your changes. You can do this for example by generating a `pull request`.

Generating a pull request

In workflows with personal public repositories, one needs to send the notification that the changes are available to co-developers, or to the maintainer, or to integration managers. The `git request-pull` command can help with this step. Given the starting point (the bottom of the revision range of interest) and the URL or the name of remote public repository, it will generate a summary of changes:

```
$ git request-pull origin/master publish
The following changes since commit
ba5807e44d75285244e1d2eacb1c10cbc5cf3935:

  Merge: strtol() + checks (2014-05-31 20:43:42 +0200)

are available in the git repository at:

  https://git.example.com/random master

Alice Developer (1):
  Support optional <count> parameter

src/rand.c |    26 ++++++-----+
1 files changed, 21 insertions(+), 5 deletions(-)
```

The pull request contains the SHA-1 of the base of the changes (which is the revision just before the first commit, in series proposed for pull), the title of the base commit, the URL and the branch of the public repository (suitable as `git pull` parameters), and the `shortlog` and `diffstat` of changes. This output can be sent to the maintainer, for example, by email.

Many Git hosting software and services include a built-in equivalent for `git request-pull` (for example, the `Create pull request` action in

GitHub).

Exchanging patches

Many larger projects (and many open-source projects) have established procedures for accepting changes in the form of patches, for example, to lower the barrier to entry for contributing. If you want to send a one-off code proposal to a project, but you do not plan to be a regular contributor, sending patches might be easier than a full collaboration setup (acquiring the permission to commit in the centralized workflow, setting up a personal public repository for the forking workflow and for similar workflows). Besides, one can generate patches with any compatible tool, and the project can accept patches no matter which version control setup they're using.

Note

Nowadays, with the proliferation of various free Git hosting services, it might be more difficult to set up an e-mail client for sending properly formatted patch emails—though services such as `submitGit` (for submitting patches to the Git project mailing list) could help.

Additionally, patches, being a text representation of changes, can be easily understood by computers and humans alike. This makes them universally appealing, and very useful for code review purposes. Many open-source projects use the public mailing list for that purpose: you can email a patch to this list, and the public can review and comment on your changes.

To generate e-mail versions of each commit series, turning them into mbox-formatted patches, you can use the `git format-patch` command, as follows:

```
$ git format-patch -M -1  
0001-Support-optional-count-parameter.patch
```

You can use any revision range specifier with this command, most commonly used is limiting by the number of commits, as in the

preceding example, or by using the double-dot revision range syntax—for example, @{u}.. (see [Chapter 2](#), *Exploring Project History*). When generating a larger number of patches, it is often useful to select a directory where to save generated patches. This can be done with the -o <directory> option. The -M option for `git format-patch` (passed to `git diff`) turns on rename detection; this can make patches smaller and easier to review.

The patch files end up looking like this:

```
From db23d0eb16f553dd17ed476bec731d65cf37cbdc Mon Sep 17
00:00:00 2001
From: Alice Developer <alice@company.com>
Date: Sat, 31 May 2014 20:25:40 +0200
Subject: [PATCH] Initialize random number generator

Signed-off-by: Alice Developer <alice@company.com>
---
random.c |    2 ++
1 files changed, 2 insertions(+), 0 deletions(-)

diff --git a/random.c b/random.c
index cc09a47..5e095ce 100644
--- a/random.c
+++ b/random.c
@@ -1,5 +1,6 @@
 #include <stdio.h>
 #include <stdlib.h>
+#include <time.h>

    int random_int(int max)
@@ -15,6 +16,7 @@ int main(int argc, char *argv[])
    int max = atoi(argv[1]);

+    srand(time(NULL));
    int result = random_int(max);
    printf("%d\n", result);
--
2.5.0
```

It is actually a complete email in the mbox format. The subject (after stripping the [PATCH] prefix) and everything up to the three-dash line --

- forms the commit message—the description of the change. To email this to a mailing list or a developer, you can use either `git send-email` or `git imap-send`. The maintainer can then use `git am` to apply the patch series, creating commits automatically; there's more about this in [Chapter 7, Merging Changes Together](#).

Note

The `[PATCH]` prefix is here to make it easier to distinguish patches from other emails. The prefix can—and often does—include additional information, such as the number in the series (set) of patches, revision of series, information about it being a work-in-progress, or the request-for-comments status, for example: `[RFC/PATCHv4 3/8]`.

You can also edit these patch files to add more information for prospective reviewers—for example, information about alternative approaches, about the differences between previous revisions of the patch (previous attempts), or a summary and/or references to the discussion on implementing the patch (for example, on a mailing list). You add such text between the `---` line and the beginning of the patch, before the summary of changes (diffstat); it will be ignored by `git am`.

Chain of trust

An important part of collaborative efforts during the development of a project is ensuring the quality of its code. This includes protection against the accidental corruption of the repository, and unfortunately also from malicious intent—a task that the version control system can help with. Git needs to ensure trust in the repository contents: your own and other developers' (including especially the canonical repository of the project).

Content-addressed storage

In [Chapter 2, Exploring Project History](#), we learned that Git uses SHA-1 hashes as a native identifier of commit objects (which represent revisions of the project, and form its history). This mechanism makes it possible to generate commit identifiers in a distributed way, taking the SHA-1 cryptographic hash function of the commit object link to the previous commit (the SHA-1 identifier of the parent commit) included.

Moreover, all other data stored in the repository (including the file contents in the revision represented by the blob objects, and the file hierarchy represented by the tree objects) also use the same mechanism. All types of object are addressed by their contents, or to be more accurate, the hash function of the object. You can say that the base of a Git repository is the content-addressed object database.

Thus Git provides a built-in **trust chain** through secure SHA-1 hashes. In one dimension, the SHA-1 of a commit depends on its contents, which includes the SHA-1 of the parent commit, which depends on the contents of the parent commit, and so forth down to the initial root commit. In the other dimension, the content of a commit object includes the SHA-1 of the tree representing the top directory of a project, which in turn depends on its contents, and these contents includes the SHA-1 of subdirectory trees and blobs of file contents, and so forth down to the individual files.

All of this allows SHA-1 hashes to be used to verify whether objects obtained from a (potentially untrusted) source are correct, and that they have not been modified since they have been created.

Lightweight, annotated, and signed tags

The trust chain allows us to verify contents, but does not verify the identity of the person that created this contents (the author and committer name are fully configurable). This is the task for GPG/PGP signatures: signed tags, signed commits, and signed merges.

Lightweight tags

Git uses two types of tags: lightweight and annotated. A **lightweight tag** is very much like a branch that doesn't change – it's just a pointer (reference) to a specific commit in the graph of revisions, though in `refs/tags/` namespace rather than in `refs/heads/` one.

Annotated tags

Annotated tags, however, involve **tag objects**. Here the tag reference (in `refs/tags/`) points to a tag object, which in turn points to a commit. Tag objects contain a creation date, the tagger identity (name and e-mail), and a tagging message. You create an annotated tag with `git tag -a` (or `--annotate`). If you don't specify a message for an annotated tag on the command line (for example, with `-m "<message>"`), Git will launch your editor so you can enter it.

You can view the tag data along with the tagged commit with the `git show` command as follows, (commit skipped):

```
$ git show v0.2
tag v0.2
Tagger: Joe R Hacker <joe@company.com>
Date:   Sun Jun 1 03:10:07 2014 -0700

random v0.2

commit 5d2584867fe4e94ab7d211a206bc0bc3804d37a9
```

Signed tags

Signed tags are annotated tags with a clear text GnuPG signature of the tag data attached. You can create it with `git tag -s` (which uses your committer identity to select the signing key, or `user.signingKey` if set), or with `git tag -u <key-id>`; both versions assume that you have a private GPG key (created, for example, with `gpg --gen-key`).

Note

Annotated or signed tags are meant for marking a release, while lightweight tags are meant for private or temporary revision labels. For this reason, some Git commands (such as `git describe`) will ignore lightweight tags by default.

Of course in collaborative workflows it is important that the signed tag is made public, and that there is a way to verify it.

Publishing tags

Git does not push tags by default: you need to do it explicitly. One solution is to individually push a tag with `git push <remote> tag <tag-name>` (here `tag <tag>` is equivalent to the longer refspec `refs/tags/<tag>:refs/tags/<tag>`); however, you can skip `tag` in most cases, here. Another solution is to push tags in mass either all the tags—both lightweight and annotated—with the use of the `--tags` option, or just all annotated tags that point to pushed commits with `--follow-tags`. This explicitness allows you to re-tag (using `git tag -f`) with impunity, if it turns out that you tagged the wrong commit, or there is a need for a last-minute fix—but only if the tag was not made public.

When fetching changes, Git automatically follows tags, downloading annotated tags that point to fetched commits. This means that downstream developers will automatically get signed tags, and will be able to verify releases.

Tag verification

To verify a signed tag, you use `git tag -v <tag-name>`. You need the signer's public GPG key in your keyring for this (imported using for example `gpg --import` or `gpg --keyserver <key-server> --recv-key <key-id>`), and of course the tagger's key needs to be vetted in your chain of trust.

```
$ git tag -v v0.2
object 1085f3360e148e4b290ea1477143e25cae995fdd
type commit
tag signed
tagger Joe Random <jrandom@example.com> 1411122206 +0200

project v0.2
gpg: Signature made Fri Jul 19 12:23:33 2014 CEST using RSA key
ID A0218851
gpg: Good signature from "Joe Random <jrandom@example.com>"
```

Signed commits

Signed tags are a good solution for users and developers to verify that the tagged release was created by the maintainer. But how do we make sure that a commit purporting to be by a somebody named Jane Doe, with the `jane@company.com` e-mail, is *actually* a commit from her? How to make it so anybody can check it?

One possible solution, available since Git version 1.7.9, is to GPG-sign individual commits. You can do this with `git commit --gpg-sign [= <keyid>]` (or `-s` in short form). The key identifier is optional—without this, Git would use your identity as the author. Note that `-s` (capital *S*) is different from `-s` (small *s*); the latter adds a *Signed-off-by* line at the end of the commit message for the Digital Certificate of Ownership.

```
$ git commit -a --gpg-sign

You need a passphrase to unlock the secret key for
user: "Jane Doe <jane@company.com>"
2048-bit RSA key, ID A0218851, created 2014-03-19

[master 1085f33] README: eol at eof
 1 file changed, 1 insertion(+), 1 deletion(-)
```

To make commits available for verification, just push them. Anyone can then verify them with the `--show-signature` option to `git log` (or `git show`), or with one of the `%Gx` placeholders in `git log --format=<format>`.

```
$ git log -1 --show-signature
commit 1085f3360e148e4b290ea1477143e25cae995fdd
gpg: Signature made Wed Mar 19 11:53:49 2014 CEST using RSA key
ID A0218851
gpg: Good signature from "Jane Doe <jane@company.com>"
```

Author: Jane Doe <jane@company.com>
Date: Wed Mar 19 11:53:48 2014 +0200

README: eol at eof

Since Git version 2.1.0, you can also use the `git verify-commit` command for this.

Merging signed tags (merge tags)

The **signed commit** mechanism, described in the previous section, may be useful in some workflows, but it is inconvenient in an environment where you push commits out early—for example, to your own public repository—and only after a while do you decide whether they are worth including in the upstream (worth sending to the main repository). This situation can happen if you follow the recommendations of [Chapter 8, Keeping History Clean](#); you know only after the fact (long after the commit was created), that the given iteration of the commit series passes code review.

You can deal with this issue by rewriting the whole commit series after its shape is finalized (after passing the review), signing each rewritten commit; or just amending and signing only the top commit. Both of those solutions would require forced push to replace old not signed history. Or you can create an empty commit (with `--allow-empty`), sign it, and push it on top of the series. But there is a better solution: requesting the pull of a signed tag (available since Git version 1.7.9).

In this workflow, you work on changes and, when they are ready, you

create and push a signed tag (tagging the last commit in the series). You don't have to push your working branch—pushing the tag is enough. If the workflow involves sending a pull request to the integrator, you create it using a tag as the end commit:

```
$ git tag -s for-maintainer
$ git request-pull origin/master public-repo 1253-for-
maintainer \
>msg.txt
```

The signed tag message is shown between the dashed lines in the pull request, which means that you may want to explain your work in the tag message when creating the signed tag. The maintainer, after receiving such pull request, can copy the repository line from it, fetching and integrating the named tag. When recording the merge result of pulling the named tag, Git will open an editor and ask for a commit message. The integrator will see the template starting with:

```
Merge tag '1252-for-maintainer'
```

```
Work on task tsk-1252
```

```
# gpg: Signature made Wed Mar 19 12:23:33 2014 CEST using RSA
key ID A0218851
# gpg: Good signature from "Jane Doe <jane@company.com>"
```

This commit template includes the commented out output of the GPG verification of the signed tag object being merged (so it won't be in the final merge commit message). The tag message helps describe the merge better.

The signed tag being pulled is *not* stored in the integrator's repository, not as a tag object. Its content is stored, hidden, in a merge commit. This is done so as to not pollute the tag namespace with a large number of such working tags. The developer can safely delete the tag (`git push public-repo --delete 1252-for-maintainer`) after it gets integrated.

Recording the signature inside the merge commit allows for after-the-fact verification with the `--show-signature` option:

```
$ git log -1 --show-signature
commit 0507c804e0e297cd163481d4cb20f3f48ceb87cb
merged tag '1252-for-maintainer'
gpg: Signature made Wed Mar 19 12:23:33 2014 CEST using RSA key
ID A0218851
gpg: Good signature from "Jane Doe <jane@company.com>""
Merge: 5d25848 1085f33
Author: Jane Doe <jane@company.com>
Date:   Wed Mar 19 12:25:08 2014 +0200
```

Merge tag 'for-maintainer'

Work on task tsk-1252

Summary

We have learnt how to use Git for collaborative development, how to work together on a project in a team. We got to know different collaborative workflows, different ways of setting up repositories for collaboration. Which one to use depends on circumstances: how large the team is, how diverse, and so on. This chapter focuses on repository-to-repository interaction; the interplay between branches and remote-tracking branches in those repositories is left for [Chapter 6, Advanced Branching Techniques](#).

We have learnt how Git can help manage information about remote repositories (remotes) involved in the chosen workflow. We were shown how to store, view, and update this information. This chapter explains how one can manage triangular workflows, in which you fetch from one repository (canonical), and push to the other (public).

We have learnt how to choose a transport protocol if the remote server offers such choice, and a few tricks such as using foreign repositories as if they were native Git repositories.

Contact with remote repositories can require providing credentials, usually the username and password, to be able to, for example, push to the repository. This chapter describes how Git can help make this part easier to use thanks to credential helpers.

Publishing your changes, sending them upstream, may involve different mechanisms, depending on the workflow. This chapter describes the push, pull request and patch-based techniques.

We have learned about the chain of trust: how to verify that a release comes from the maintainer, how to sign your work so that the maintainer can verify that it comes from you, and how the Git architecture helps with this.

The two following chapters will expand the topic of collaboration:

[Chapter 6](#), *Advanced Branching Techniques*, will explore relations between local branches and branches in a remote repository, and how to set up branches for collaboration, while [Chapter 7](#), *Merging Changes Together*, will talk about the opposite issue—how to join the results of parallel work.

Chapter 6. Advanced Branching Techniques

The previous chapter, *Collaborative Development with Git*, described how to arrange teamwork, focusing on repository-level interactions. In that chapter, you learned about various centralized and distributed workflows, and their advantages and disadvantages.

This chapter will go deeper into the details of collaboration in a distributed development. It would explore the relations between local branches and branches in remote repositories. It will introduce the concept of remote tracking branches, branch *tracking*, and *upstream*. This chapter will also teach us how to specify the synchronization of branches between repositories, using `refsspecs` and push modes.

You will also learn branching techniques: how branches can be used to prepare new releases and to fix bugs. You will learn how to use branches in such way so that it makes it easy to select which features go into the next version of the project.

In this chapter, we will cover the following topics:

- Different kinds of branches, both long-lived and short-lived, and their purpose
- Various branching models, including topic branch-based workflow
- Release engineering for different branching models
- Using branches to fix a security issue in more than one released version
- Remote-tracking branches and `refsspecs`, the default remote configuration
- Rules for fetching and pushing branches and tags
- Selecting a push mode to fit chosen collaboration workflow

Types and purposes of branches

A **branch** in a version control system is a active parallel line of development. They are useful, as we will see, to isolate and separate different types of work. For example, branches can be used to prevent your current work on a feature in progress from interfering with the management of bug fixes.

A single Git repository can have an arbitrary number of branches. Moreover, with a distributed version control system, such as Git, there could be many repositories (forks) for a single project, some public and some private; each repository will have its own local branches.

Before examining how the collaboration between repositories looks like at the branch level, we need to know what types of branches we would encounter in local and remote repositories. Let's now talk about how these branches are used and examine why people would want to use multiple branches in a single repository.

Note

A bit of history: a note on the evolution of branch management

Early distributed version control systems used one branch per repository model. Both **Bazaar** (then Bazaar-NG) and **Mercurial** documentation, at the time when they begin their existence, recommended to clone the repository to create a new branch.

Git, on the other hand, had good support for multiple branches in a single repository almost from the start. However, at the beginning, it was assumed that there would be one central multibranch repository interacting with many single-branch repositories (see, for example, the legacy `.git/branches` directory to specify URLs and fetch branches, described in the `gitrepository-layout(7)` man page), though with Git it was more about defaults than capabilities.

Because branching is cheap in Git (and merging is easy), and collaboration is quite flexible, people started using branches more and more, even for solitary work. This led to the wide use of extremely

useful topic branch workflow.

There are many reasons for keeping a separate line of development, thus there are many kinds of branches. Different types of branches have different purposes. Some branches are long-lived or even permanent, while some branches are short-lived and expected to be deleted after their usefulness ends. Some branches are intended for publishing, some are not.

Long-running, perpetual branches

Long-lived or permanent branches are intended to last (indefinitely or, at least, for a very long time).

From the collaboration point of view, a long-lived branch can be expected to be there when you are next updating data or publishing changes. This means that one can safely start their own work basing it on (forking it from) any of the long-lived branches in the remote repository, and be assured that there should be no problems with integrating that work.

Also, what you can find in public repositories are usually only long-lived branches. In most cases, these branches should never rewind (the new version is always a descendant of the old versions). There are some special cases here though; there can be branches that are rebuilt after each new release (requiring forced fetch at that time), and there can be branches that do not fast forward. Each such case should be explicitly mentioned in the developer documentation to help avoiding unpleasant surprises.

Integration, graduation, or progressive-stability branches

One of the uses of branches is to separate ongoing development (which can include temporarily some unstable code) from maintenance work (where you are accepting only bug fixes). There are usually a few of such branches. The intent of each of these branches is to *integrate* the development work of the respective degree of stability, from

maintenance work, through stable, to unstable or development work.

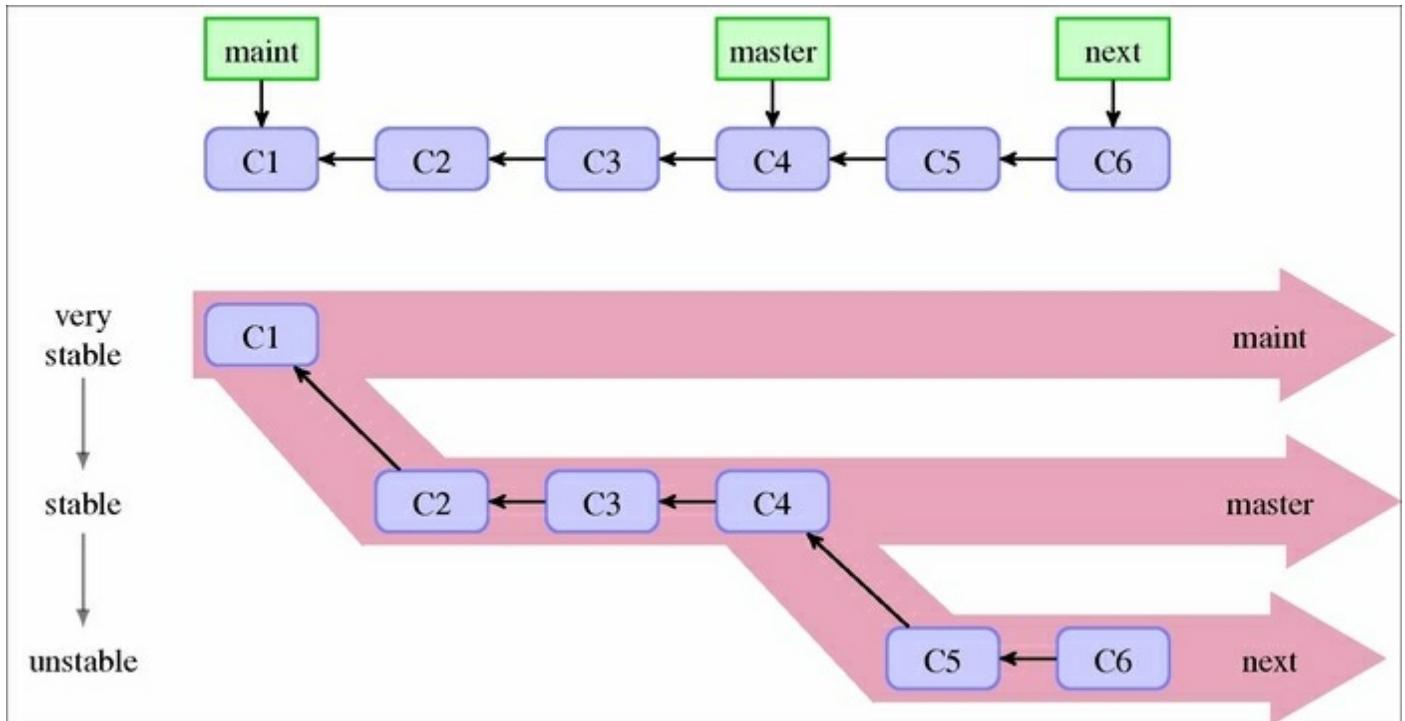


Fig 1. A linear view and a "silo" view of the progressive-stability branches. In the linear view, the stable revisions are further down the line in your commit history, and the cutting-edge unstable work is further up the history. Alternatively, we can think of branches as work silos, where work goes depending on the level of the stability (graduation) of changes.

These branches form a hierarchy with a decreasing level of graduation or stability of work, as shown in *Fig 1*. Note that, in real development, progressive-stability branches would not keep this simple image exactly as it is shown. There would be new revisions on the branches after the forking points. Nevertheless, the overall shape would be kept the same, even in the presence of merging.

The rule is to always merge more stable branches into less stable ones, that is, *merge upwards*, which will preserve the overall shape of branch silos (see also *Fig 2* in the *Graduation, or progressive-stability branches workflow* section of this chapter). This is because merging means including all the changes from the merged branch. Therefore,

merging a less stable branch into a more stable one would bring unstable work to the stable branch, violating the purpose and the contract of a stable branch.

Often, we see the graduation branches of the following levels of stability:

- `maint` or `maintenance` of the `fixes` branch, containing only bug fixes to the last major release; minor releases are done with the help of this branch.
- The `master` or `trunk`, or `stable` branch, with the development intended for the next major release; the tip of this branch should be always in the production-ready state.
- `next` or `devel`, `development`, or `unstable`, where the new development goes to test whether it is ready for the next release; the tip can be used for nightly builds.
- `pu` or `proposed` for the proposed updates, which is the integration testing branch meant for checking compatibility between different new features.

Having multiple long-running branches is not necessary, but it's often helpful, especially in very large or complex projects. Often in operations, each of levels of stability corresponds to its own platform or deployment environment; giving a branch per platform.

Per-release branches and per-release maintenance

Preparing for the new release of a project can be a lengthy and involved process. Per-release branches can help with this. The release branch is meant for separating the ongoing development from preparing the new release. It allows other developers to continue working on writing new features and on integration testing, while the quality assurance team with the help of the release manager takes time to test and stabilize the release candidate.

After creating a new release, keeping such per-release branches allows us to support and maintain older released versions of the software. At these times, such branches work as a place to gather bug fixes (for their

software versions) and create minor releases.

Not all the projects find utilizing per-release branches necessary. You can prepare a new release on the stable-work graduation branch, or use a separate repository in place of using a separate branch. Also, not all the projects require providing support for more than the latest version.

This type of branches is often named after the release it is intended for, for example, having names such as `release-v1.4`, or `v1.4.x` (it better not have the same name as tag for release, though).

Hotfix branches for security fixes

Hotfix branches are like release branches, but for unplanned releases. Their purpose is to act upon the undesired state of a live production or a widely deployed version, usually to resolve some critical bug in the production (usually a severe security bug). This type of branches can be considered a longer lived equivalent of the bugfix topic branches (see the *Bugfix branches* section of this chapter).

Per-customer or per-deployment branches

Let's say that some of your project's customers require a few customization tweaks, since they do things differently. Or perhaps, there are some deployment sites that have special requirements. Suppose that these customizations cannot be done by simply changing the configuration. You would then need to create separate the lines of development for these customers or customizations.

But you don't want these lines of development to remain separate. You expect that there will be changes that apply to all of them. One solution is to use one branch for each customization set, per customer or per deployment. Another would be to use separate repositories. Both solutions help maintain parallel lines of development and transfer changes from one line to another.

Automation branches

Say that you are working on a web application and you want to automate its deployment using a version control system. One solution would be to set up a daemon to watch a specific branch (for example the one named 'deploy') for changes. Updating such branch would automatically update and reload the application.

This is, of course, not the only possible solution. Another possibility would be to use a separate deploy repository and set up hooks there, so push would trigger refreshing of the web application. Or, you could configure a hook in a public repository so that push to a specific branch triggers redeployment (this mechanism is described in [Chapter 11, Git Administration](#)).

These techniques can be used also for **continuous integration (CI)**; instead of deploying the application, pushing it into a specific branch would trigger the running of test suite (the trigger could be creating a new commit on this branch or merging into it).

Mob branches for anonymous push access

Having a branch in a remote repository (on server) with special treatment on push, is a technique that has many uses, including helping to collaborate. It can be used to enable *controlled* anonymous push access for a project.

Let's assume that you want to allow random contributors to push into the central repository. You would want, however, to do this in a managed way: one solution is to create a special `mob` branch or a `mob/*` namespace (set of branches) with relaxed access control.

You can find how to set this up in [Chapter 11, Git Administration](#).

The orphan branch trick

All the types of branches described up to this point differed in their purpose and management. However, from the technical point of view (from the point of view of the graph of commits), they all look the same. This is not the case with the so-called *orphan* branches.

The orphan branch is a parallel disconnected (orphaned) line of development, sharing no revisions with the main history of a project. It is a reference to a disjoint subgraph in the DAG of revisions, without any intersection with the main DAG graph. In most cases, their checkout is also composed of different files.

Such branches are sometimes used as a trick to store tangentially related contents in a single repository, instead of using separate repositories. (When using separate repositories to store related contents, one might want to use some naming convention to denote this fact, for example a common prefix.) They can be used to:

- Store the project's web page files. For example, GitHub uses a branch named `gh-pages` for the project's pages.
- Store generated files, when the process of creating them requires some nonstandard toolchain. For example, the project documentation can be stored in `html`, `man`, and `pdf` orphan branches (the `html` branch can be also used to deploy the documentation). This way the user can get it without needing to install its toolchain.
- Store the project TODO notes (for example in the `todo` branch), perhaps together with storing there some specialized maintainer tools (scripts).

You can create such branch with `git checkout --orphan <new branch>`, or by pushing into (or fetching into) a specific branch from a separate repository, as follows:

```
$ git fetch repo-htmldocs master:html
```

Note

Creating an orphan branch with `git checkout --orphan` does not technically create a branch, that is, it does not make a new branch reference. What it does is point the symbolic reference `HEAD` to an unborn branch. The reference is created after the first commit on a new orphan branch.

That is why there is no option to create an orphan branch for `git`

branch command.

Short-lived branches

While long-lived branches stay forever, short-lived or temporary branches are created to deal with single issues, and are usually removed after dealing with said issue. They are intended to last only as long as the issue is present. Their purpose is time-limited.

Because of their provisional nature, they are usually present only in the local private repository of a developer or integration manager (maintainer), and are not pushed to public distribution repositories. If they appear in public repositories, they are there only in a public repository of an individual contributor (see the blessed repository workflow in [Chapter 5, Collaborative Development with Git](#)), as a target for a pull request.

Topic or feature branches

Branches are used to separate and gather together different subsets of development efforts. With easy branching and merging, we can go further than creating a branch for each stability level, as described earlier. We can create a separate branch for each separate issue.

The idea is to make a new branch for each topic, that is, a feature or a bug fix. The intent of this type of branch is both to gather together subsequent development steps of a feature (where each step – a commit – should be a self contained piece, easy to review) and to isolate the work on one feature from the work on other topics. Using a feature branch allows topical changes to be kept together and not mixed with other commits. It also makes it possible for a whole topic to be dropped (or reverted) as a unit, be reviewed as a unit, and be accepted (integrated) as a unit.

The end goal for the commits on a topic branch is to be included in a released version of a product. This means that, ultimately, the short-lived topic branch is to be merged into the long-lived branch which is

gathering stable work, and to be deleted. To make it easier to integrate topic branches, the recommended practice is to create such branches by forking off the oldest, the most stable integration branch that you will eventually merge it into. Usually, this means creating a branch from the stable-work graduation branch. However, if a given feature does depend on a topic not yet in the stable line, you need to fork off the appropriate topic branch containing the dependency you need.

Note that if it turns out that you forked off the wrong branch, you can always fix it by rebasing (see [Chapter 7, Merging Changes Together](#), and [Chapter 8, Keeping History Clean](#)), as topic branches are not public.

Bugfix branches

We can distinguish a special case of a topic branch whose purpose is fixing a bug. Such branch should be created starting from the oldest integration branch it applies to (the most stable branch that contains the bug). This usually means forking off the maintenance branch, or off the divergence point of all the integration branches, rather than the tip of the stable branch. A bugfix branch's goal is to be merged into relevant long-lived integration branches.

Bugfix branches can be thought of as a short-lived equivalent of a long-lived hotfix branch.

Using them is a better alternative to simply committing fixes on the maintenance branch (or another appropriate integration branch).

Detached HEAD – the anonymous branch

You can think of the detached `HEAD` state (described in [Chapter 3, Developing with Git](#)) as the ultimate in temporary branches—so temporary that it even doesn't have a name. Git uses such anonymous branches automatically in a few situations, for example, during bisection and rebasing.

Because, in Git, there is only one anonymous branch and it must always

be the current branch, It is usually better to create a true temporary branch with a temporary name; you can always change the name of the branch later.

One possible use of the detached HEAD is for proof of concept work. You, however, need to remember to set the name of the branch if the changes turn out to be worthwhile (or if you need to switch branches). It is easy to go from an anonymous branch to a named branch. You simply need to create a new branch from the current detached HEAD state.

Branching workflows and release engineering

Now that we know what types of branches are there and what their purposes are, let's examine how branches are used. Note that different situations call for different use of branches. For example, smaller projects are better suited for simpler branching workflows, while larger projects might need more advanced ones.

We will now describe here how to use different standard workflows. Each workflow is distinguished by the various types of branches it uses (the types described earlier in this chapter). In addition to getting to know how the ongoing development looks like for a given workflow, we would also see what to do at the time of the new release (major and minor, where relevant). Among others, we will find out what happens then to branches used in the chosen workflow.

The release and trunk branches workflow

One of the simplest workflows is to use just a single integration branch. Such branches are sometimes called **the trunk**; in Git, it would usually be the `master` branch (it is the default branch when creating a repository). In a pure version of this workflow, one would commit everything on the said branch, at least, during the normal development stage. This way of working comes from the times of centralized version control, when branching and especially merging was more expensive and people avoided branch-heavy workflows.

Note

In more advanced versions of this workflow, one would also use topic branches, one short-lived branch per feature, and merge them into the trunk, instead of committing directly on it (see *Fig 3*).

In this workflow, we create the new release branch out of trunk when

deciding to cut the new major release. This is done to avoid the interference between stabilizing for release and ongoing development work. The rule is that all the stabilization work goes on the release branch, while all the ongoing development goes to the trunk. **Release candidates** are cut (tagged) from the release branch, as is the final version of a release.

The release branch for a given version can be later used to gather bug fixes, and to cut minor releases from it.

The disadvantage of such simple workflow is that during development, we often get in an unstable state. In this case, it could be hard to come up with a good starting point, stable enough to start working on creating a new release. An alternative solution is to create revert commits on the release branch, undoing the work that is not ready. But it can be a lot of work and it would make the history of a project hard to follow.

Another difficulty with this workflow is that the feature that looks good at the first glance might show problems later in use. This is something this workflow has trouble dealing with. If it turns out during development that some feature created in multiple commits feature is not a good idea, reverting it can be difficult. This is true especially if its commits are spread across the timeline (across the history).

Moreover, the trunk and release branch workflow does not provide any inherent mechanism for finding bad interactions between different features, that is, for the integration testing.

In spite of these problems, this simple workflow can be a good fit for a small team.

The graduation, or progressive-stability branches workflow

To be able to provide the stable line of the product and to be able to test it in practice as a kind of floating beta version, one needs to separate work that is stable from the work that is ongoing and might destabilize

code. That's what graduation branches are for: to integrate revisions with different degrees of maturation and stability (this type of long-running branches is also called **integration** branches or **progressive-stability** branches). See *Fig 1* of the *Integration, or graduation, or progressive-stability branches* section in this chapter, which shows a graph view and a silo view of a simple case with progressive-stability branches and linear history. Let's call the technique that utilizes mainly (or only) this type of branches **the graduation branches workflow**.

Besides keeping stable and unstable development separate sometimes, there is also a need for an ongoing maintenance. If there is only one version of the product to support, and the process of creating a new release is simple enough, one can also use the graduation-type branch for this.

Note

Here, *simple enough* means that one can just create the next major release out of the stable branch.

In such situation, one would have at least three integration branches. There would be one branch for the ongoing maintenance work (containing only bug fixes to the last version), to create minor releases. One branch for stable work to create major releases; this branch can also be used for nightly stable builds. And last, one branch for ongoing development, possibly unstable.

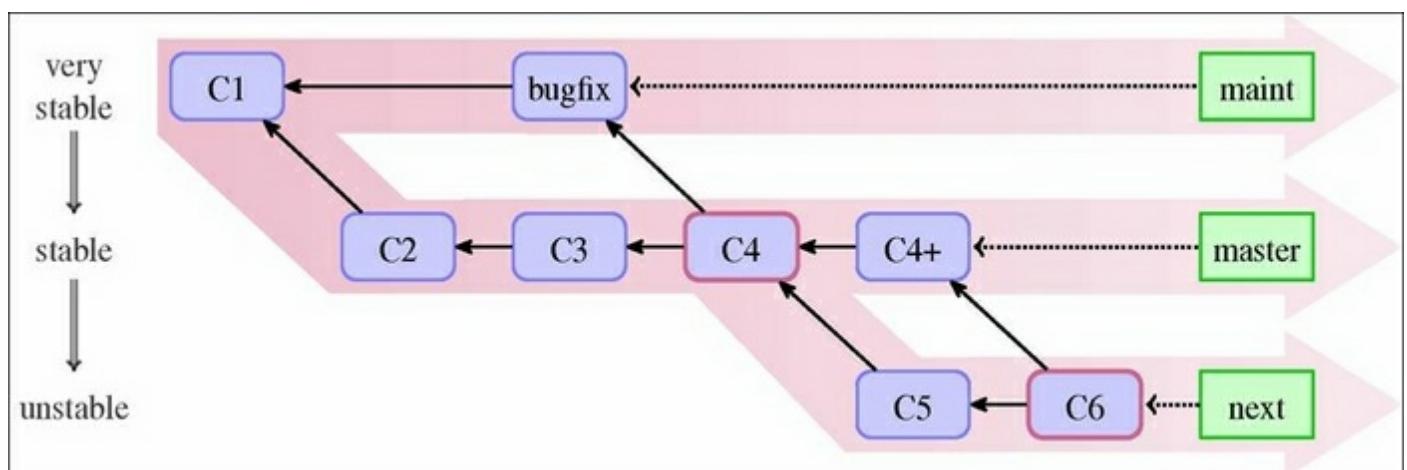


Fig 2. The graduation or progressive-stability branches workflow. You

should never merge a less stable branch into more stable one, as merging would bring all the unstable history.

You can use this workflow as it is, with only graduation branches, and no other types of branches. You commit bug fixes on the maintenance branch and merge it into the stable branch and development branch, if necessary. You create revisions with the well-tested work on the stable branch, merging it into the development branch when needed (for example, if the new work depends on them). You put the work in progress, possibly unstable, on the development branch. During normal development, you never merge less stable into more stable branches, otherwise you would decrease their stability. It is always more stable into less stable, as represented in *Fig 2*.

This, of course, requires that you know upfront whether the feature that you are working on should be considered stable or unstable. There is also an underlying assumption that different features work well together from the start. One would expect in practice, however, that each piece of the development matures from the proof of concept, through being a work in progress during possibly several iterations, before it stabilizes. This problem can be solved with the workflow involving use of topic branches, which will be described next.

In the pure graduation branches workflow, one would create minor releases (with bug fixes) out of the maintenance branch. Major releases (with new features) are created out of the stable-work branch. After a major release, the stable-work branch is merged into the maintenance branch to begin supporting the new release that was just created. At this point also, an unstable (development) branch can be merged into a stable one. This is the only time when merging upstream, which means merging less stable branches into more stable branches, should be done.

The topic branches workflow

The idea behind the topic branches workflow is to create a separate

short-lived branch for each topic, so that all the commits belonging to a given topic (all the steps in its development) are kept together. The purpose of each topic branch is a development of the new feature, or a creation of a bug fix.

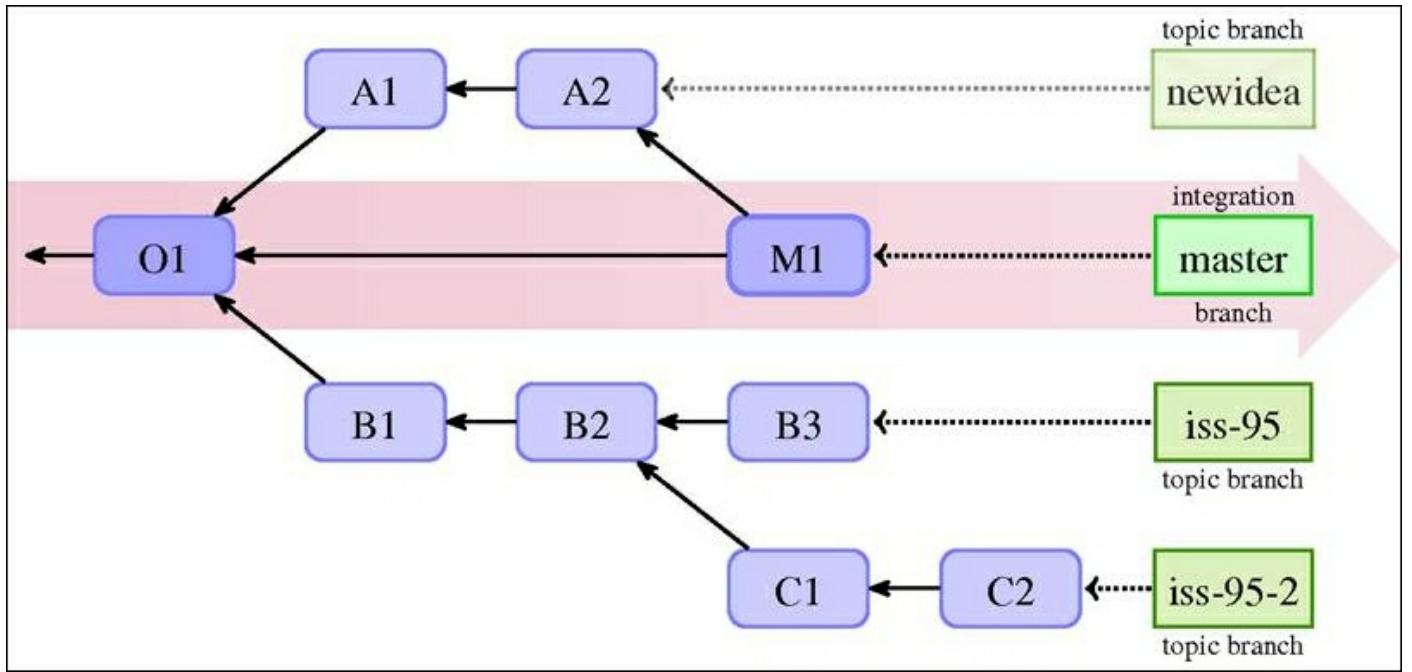


Fig 3. The topic branches workflow with one integration branch (master) and three topic or feature branches. Among the topic branches, there is one (namely, newidea) already merged in the integration branch and one (iss-95-2) dependent on the feature developed in the other feature branch (iss-95 here).

In the **topic branches workflow** (also called the **feature branches workflow**), you have at least two different types of branches. First, there needs to be at least one permanent (or just long-lived) *integration branch*. This type of branches is used purely for merging. Integration branches are public.

Second, there are separate short-lived temporary *feature branches*, each intended for the development of a topic or the creation of a bug fix. They are used to carry all the steps, and only the steps required in the development of a feature or a fix; a unit of work for a developer. These branches can be deleted after the feature or the bug fix is merged. Topic

branches are usually private and are often not present in public repositories.

When a feature is ready for review, its topic branch is often rebased to make integration easier, and optionally to make history more clear. It is then sent for review as a whole. The topic branch can be used in a pull request, or can be sent as a series of patches (for example, using `git format-patch` and `git send-email`). It is often saved as a separate topic branch in a maintainer's working repository (for example, `git am --3way` if it was sent as patches) to help in examining and managing it.

Then, the integration manager (the maintainer in the blessed repository workflow, or simply another developer in the central repository workflow) reviews each topic branch and decides whether it is ready for inclusion in selected integration branch. If it is, then it will get merged in (perhaps, with the `--no-ff` option).

Graduation branches in a topic branch workflow

The simplest variant of the topic branches workflow uses only one integration branch. Usually, however, one would combine the graduation branches workflow with topic branches.

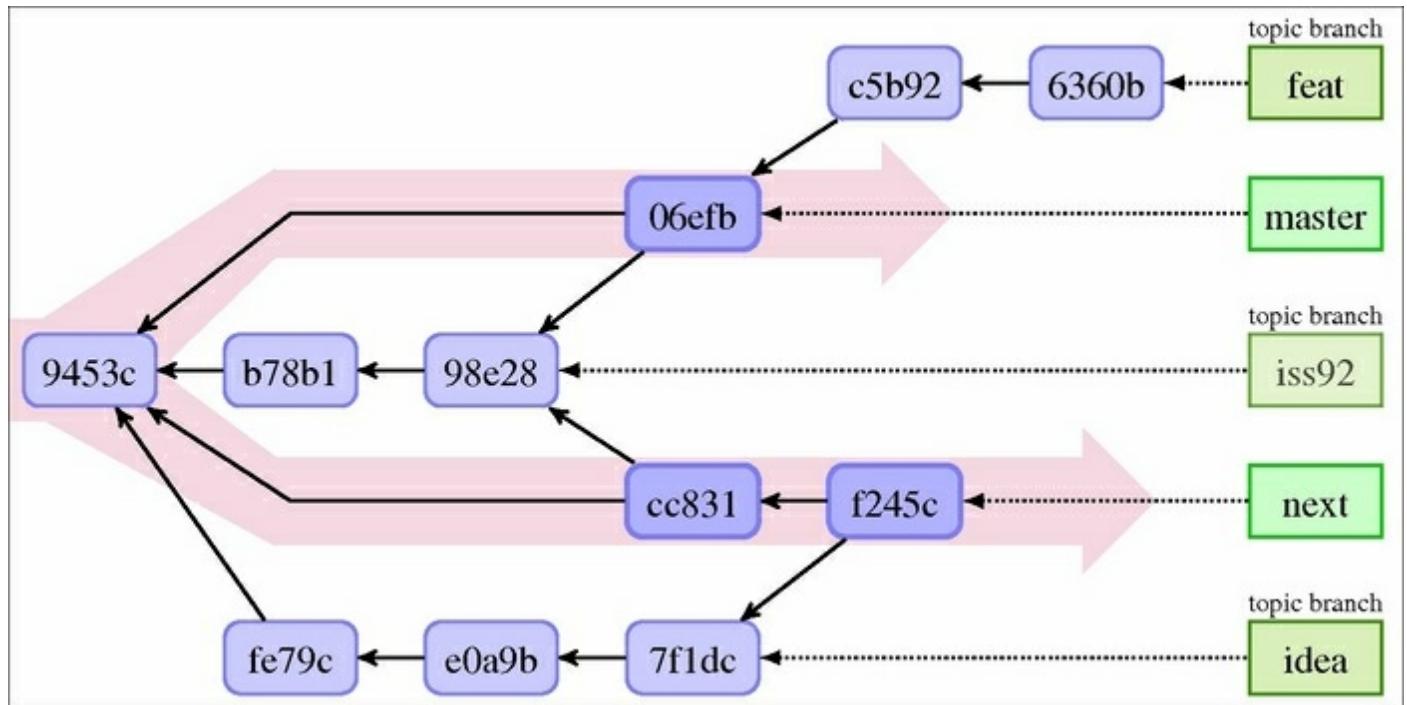


Fig 4. The topic branches workflow with two graduation branches.

Among topic branches, there is one (`iss92`) that is considered stable enough to be merged into both the `next` (unstable) and `master` (stable) graduation branches. One (`idea`) that got merged into `next` for testing and one (`feat`) just created from `master`.

In this often used variant, the feature branch is started from the tip of a given stable branch (usually) or from the last major release, unless the branch requires some other feature. In the last case, the branch needs to be forked from (created from) the topic branch it depends on, such as the `feat` branch in *Fig 4*. Bugfix topic branches are created on top of the maintenance branch.

When the topic is considered done, it is first merged into the development-work integration branch (for example, `next`) to be tested. For example, in *Fig 4*, topic branches `idea` and `iss92` are both merged into `next`, while `feat` is not considered ready yet. Adventurous users can use builds from given unstable branch to exercise the feature, though they better take into the account the possibility of crashes and data loss.

After this examination, when the feature is considered to be ready to be included in the next release, it is merged into the stable-work integration branch (for example, `master`). *Fig 4* includes one such branch: `iss92`. At this point, after merging it into the stable integration branch, the topic branch can be deleted.

Using a feature branch allows topical revision to be kept together and not mixed with other commits. The topic branch workflow allows for the easy undoing of topic as a whole, and for removing of all bad commits together (removing a series of commits as a whole unit), instead of using a series of reverts.

If the feature turns out to be not ready, it is simply not merged into the stable branch, and it remains present only in the development-work branch. If we, however, realize too late that it was not ready, after the topic was merged into the stable branch, we would need to revert the

merge. This is a slightly more advanced operation than reverting a single commit, but it is less troublesome than reverting commits one, by one while ensuring that all the commits get correctly reverted. Problems with reverting merges will be covered in [Chapter 8, Keeping History Clean](#).

The workflow for topic branches containing bugfixes is similar. The only difference is that one needs to consider into which of integration branches the bugfix branch is to be merged into. This, of course, depends on the situation. Perhaps the bugfix applies only to the maintenance branch, because it was accidentally fixed by a new feature in the stable-work and development-work branches; then, it is merged only to this branch. Perhaps, the bug applies only to the stable-work and development-work branches, because it is about the feature that was not present in the previous version, thus the maintenance branch is excluded from being merged into.

Using a separate topic branch for bug fixing, instead of committing bugfix directly, has an additional advantage. It allows us to easily correct the misstep, if it turns out after the fact that the fix applies to more branches than we thought.

For example, if it turns out that the fix needs to be applied also to the maintained version and not only to the current work, with the topic branch you can simply merge the fix into additional branches. This is not the case if we were to commit the fix directly on the stable branch. In the latter situation, you cannot use merging, as it would destabilize the maintenance branch. You would need to copy the revision with the fix, by cherry-picking it from the branch it was committed on into the maintenance branch (see [Chapter 7, Merging Changes Together](#) for detailed description of this operation). But it means that duplicated commits; additionally cherry-picked commits can sometimes interact wrongly with the act of merging.

The topic branches workflow also allows us to check whether the features conflict with each other, and then fix them as necessary. You can simply create a throw-away integration branch and merge into it topic branches containing these features, to test the interaction between

them. You can even publish such branches meant for integration testing (named `proposed-updates` or just `pu` for example) to allow for other developers to examine the works in progress. You should however state explicitly in the developer documentation that said branch should not be used as a basis to work on, as it is recreated each time from scratch.

Branch management for a release in a topic branch workflow

Let's assume that we are using three graduation (integration) branches: `maint` for maintenance work on the last release, `master` for stable work, `next` for development.

The first thing that the maintainer (the release manager) needs to do before creating a new release is to verify that `master` is a superset of `maint`, that is, all the bugs are fixed also in the version considered for the next release. You can do this by checking whether the following command gives an empty output (see [Chapter 2, Exploring Project History](#)):

```
$ git log master..maint
```

If the preceding command show some unmerged commits, the maintainer needs to decide what to do with them. If these bug fixes don't break anything, he/she can simply merge `maint` into `master` (as it is merging the more stable branch into the less stable one).

Now that the maintainer knows that `master` is a superset of `maint`, he/she can create the new release from remote `master` by tagging it, and then pushing just created tag to the distribution point (to the public repository), for example with the following:

```
$ git tag -s -m "Foo version 1.4" v1.4 master
$ git push origin v1.4 master
```

The preceding command assumed that the public repository of the `Foo` project is the one described by the `origin`, and that we use the double-digit version for major releases (following the semantic versioning

specification: <http://semver.org/>).

Note

If the maintainer wants to support more than one older version, he or she would need to copy an old maintenance branch, as the next step would be to prepare it for maintaining just released revision:

```
$ git branch maint-1.3.x maint
```

Then, the maintainer updates `maint` to the new release, advancing the branch (note that step one ensured that `maint` was a subset of `master`):

```
$ git checkout maint
$ git merge --ff-only master
```

If the second command fails, it means that there are some commits on the branch `maint` that are not present in `master`, or to be more exact that `master` is not a strict descendant of `maint`.

Because we usually consider features for inclusion in `master` one by one, there might be some topic branches that are merged into `next`, but they were abandoned before they were merged into `master` (or they are not merged because they were not ready.) This means that though the `next` branch contains a superset of topic branches that compose the `master` branch, `master` is not necessarily the ancestor of `next`.

That's why advancing the `next` branch after a release can be more complicated than advancing the `maint` branch. One solution is to rewind and rebuild the `next` branch:

```
$ git checkout next
$ git reset --hard master
$ git merge ai/topic_in_next_only_1...
```

You can find unmerged topics to be merged to rebuild `next` with:

```
$ git branch --no-merged next
```

After creating the release following rebuilding of `next`, other developers

would have to force fetch the `next` branch (see the next section), as it would not fast-forward if it is not already configured to force fetch:

```
$ git pull
From git://git.example.com/pub/scm/project
 62b553c..c2e8e4b  maint      -> origin/maint
  a9583af..c5b9256  master     -> origin/master
+ 990ffec...cc831f2  next      -> origin/next  (forced update)
```

Notice the forced update for the `next` branch here.

Git-flow – a successful Git branching model

One can see that the more advanced version of the topic branching workflow builds on top of the graduation branch's one. In some cases, even more involved branching model might be necessary, utilizing more types of branches: graduation branches, release branches, hotfix branches, and topic branches. Such model is sometimes called `gitflow` or `git-flow`.

This development model uses two main long-running graduation branches to separate the production-ready stable state from the work involved with integration of the latest delivered ongoing development. Let's call these branches for example `master` (stable work) and `develop` (gathers changes for the next release). The latter can be used for nightly builds. These two integration branches have an infinite lifetime.

These branches are accompanied in this workflow by supporting branches, namely, feature branches, release branches, and hotfix branches.

Each new feature is developed on a topic branch (such branches are sometimes called feature branch), named after a feature. Such branches are forked off the tip of either the `develop` or `master` branch, depending on the details of the workflow and the requirements of the feature in question. When work on a feature is finished, its topic branch is merged, with the `--no-ff` option (so that there is always a merge commit where a feature can be described), into `develop` for integration testing. When they

are ready for the next release, they are merged into the `master` branch. A topic branch exists only as long as a feature is in development, and are deleted when merged (or when abandoned).

The purpose of a release branch is twofold. When created, the goal is to prepare a new production release. This means doing last minute clean-up, applying minor bug fixes, and preparing metadata for a release (for example, version numbers, release names, and so on). All but the last should be done using topic branches; preparing metadata can be done directly on the release branch. This use of the release branch allows us to separate the quality assurance for the upcoming release from the work developing features for the next big release.

Such release branches are forked off when the stable state reflects, or is close to, the desired state planned for the new release. Each such branch is named after a release, usually something such as `release-1.4` or `release-v1.4.x`. One would usually create a few release candidates from this branch (tagging them `v1.4-rc1` and so on) before tagging the final state of the new release (for example, `v1.4`).

The release branch might exist only until the time the project release it was created for is rolled out, or it might be left to gather maintenance work: bug fixes for the given release (though, usually, maintenance is done only for a few latest versions or the most popular versions). In the latter situation, it replaces the `maint` branch of other workflows.

Hotfix branches are like release branches, but for an unplanned release usually connected with fixing serious security bugs. They are usually named `hotfix-1.4.1` or something similar. A hotfix branch is created out of an old release tag if the respective release (maintenance) branch does not exist. The purpose of this type of branches is to resolve critical bugs found in a production version. After putting a fix on such branches, the minor release is cut (for each such branch).

Fixing a security issue

Let's examine another situation now. How can we use branches to

manage fixing a bug, for example, a security issue. This requires a slightly different technique than an ordinary development.

As explained in *Topic branches workflow*, while it is possible to create a bugfix commit directly on the most stable of the integration branches that is affected by the bug, it is usually better to create a separate topic branch for the bugfix in question.

You start by creating a bugfix branch forking from the oldest (most stable) integration branch the fix needs to be applied to, perhaps even at the branching point of all the branches it would apply to. You put the fix (perhaps, consisting of multiple commits) on the branch that you have just created. After testing it, you simply merge the bugfix branch into the integration branches that need the fix.

This model can be also used to resolve conflicts (dependencies) between branches at an early stage. Let's assume that you are working on some new feature (on a topic branch), which is not ready yet. While writing it, you have noticed some bugs in the development version and you know how to fix them. You want to work on top of the fixed state, but you realize that other developers would also want the bugfix. Committing the fix on top of the feature branch takes the bugfix hostage. Fixing the bug directly on an integration branch has a risk of forgetting to merge the bugfix into the feature in progress.

The solution is to create a fix on a separate topic branch and to merge it into both the topic branch for the feature being developed, and into the test integration branch (and possibly the graduation branches).

Note

You can use similar techniques to create and manage some features that are requested by a subset of customers. You need to simply create a separate topic branch for each such feature and merge it into the individual, per customer branches.

The matter complicates a bit if there is security involved. In the case of a severe security bug, you would want to fix it not only in the current

version, but also in all the widely used versions.

To do this, you need to create a hotfix branch for various maintenance tracks (forking it from the specified version):

```
$ git checkout -b hotfix-1.9.x v1.9.4
```

Then, you need to merge the topic branch with the fix in question into the just created hotfix branch, to finally create the bugfix release:

```
$ git merge CVE-2014-1234
$ git tag -s -m "Project 1.9.5" v1.9.5
```

Interacting with branches in remote repositories

We see that having many branches in a single repository is very useful. Easy branching and merging allows for powerful development models, which are utilizing advanced branching techniques, such as topic branches. This means that remote repositories will also contain many branches. Therefore, we have to go beyond just the repository to the repository interaction, which was described in [Chapter 5, Collaborative Development with Git](#). We have to consider how to interact with multiple branches in the remote repositories.

We also need to think about how many local branches in our repository relate to the branches in the remote repositories (or, in general, other refs). The other important knowledge is how the tags in the local repository relate to the tags in other repositories.

Understanding the interaction between repositories, the branches in these repositories, and how merge changes (in [Chapter 7, Merging Changes Together](#)) is required to truly master collaboration with Git.

Upstream and downstream

In software development, the **upstream** refers to a direction toward the original authors or the maintainers of the project. We can say that the repository is upstream from us if it is closer (in the repository-to-repository steps) to the blessed repository—the canonical source of the software. If a change (a patch or a commit) is accepted upstream, it will be included either immediately or in a future release of an application, and all the people **downstream** would receive it.

Similarly, we can say that a given branch in a remote repository (the maintainer repository) is an **upstream branch** for given local branch, if changes in that local branch are to be ultimately merged and included in the remote branch.

Note

A quick reminder: the upstream repository and the upstream branch in the said remote repository for a given branch are defined, respectively, by the `branch.<branchname>.remote` and `branch.<branchname>.merge` configuration variables. The upstream branch can be referred to with the `@{upstream}` or `@{u}` shortcut.

The upstream is set while creating a branch out of the remote-tracking branch, and it can be modified using either `git branch --set-upstream-to` or `git push --set-upstream`.

The upstream branch does not need to be a branch in the remote repository. It can be a local branch, though we usually say then that it is a **tracked branch** rather than saying that it is an upstream one. This feature can be useful when one local branch is based on another local branch, for example, when a topic branch is forked from other topic branch (because it contains the feature that is a prerequisite for the latter work).

Remote-tracking branches and refspec

While collaborating on a project, you would be interacting with many repositories (see the *Collaborative Development With Git* section of this chapter). Each of these remote (public) repositories you are interacting with will have their own notion of the position of the branches. For example, the `master` branch in the remote repository `origin` needs not to be at the same place as your own local `master` branch in your clone of the repository. In other words, they need not point to the same commit in the DAG of revisions.

Remote-tracking branches

To be able to check the integration status, for example, what changes are there in the `origin` remote repository that are not yet in yours, or what changes did you make in your working repository that you have not yet published, you need to know where the branches in the remote

repositories are (well, where they were the last time you contacted these repositories). This is the task of remote-tracking branches—the references that track where the branch was in the remote repository.

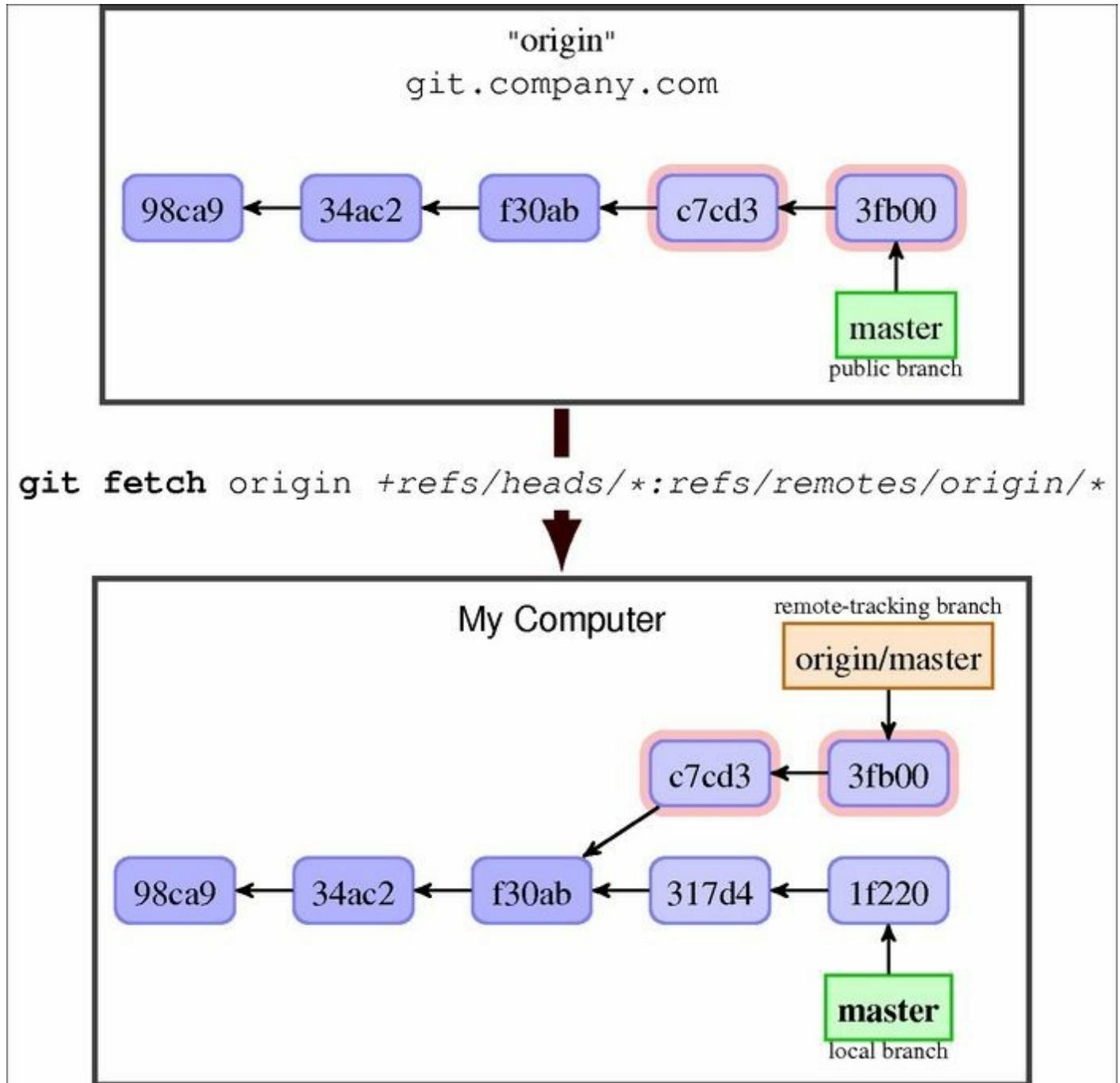


Fig 5: Remote-tracking branches. The branch `master` in remote `origin` is fetched into the remote-tracking branch `origin/master` (full name `refs/remotes/origin/master`). Grayed out text in the fetch command denotes the default implicit parameters.

To track what happens in the remote repository, remote-tracking branches are updated automatically; this means that you cannot create new local commits on top of them (as you would lose these commits during update). You need to create the local branch for it. This can be done, for example, with simply `git checkout <branchname>`, assuming that the local branch with the given name does not already exist. This command creates a new local branch out of the remote branch `<branchname>` and sets the upstream information for it.

Refspec – remote to local branch mapping specification

As described in [Chapter 2](#), *Exploring Project History*, local branches are in the `refs/heads/` namespace, while remote-tracking branches for a given remote are in the `refs/remotes/<remote name>/` namespace. But that's just the default. The `fetch` (and `push`) lines in the `remote.<remote name>` configuration describe the mapping between branches (or refs in general) in the remote repository and the remote-tracking branches (or other refs) in the local repository.

This mapping is called **refs**pec; it can be either explicit, mapping branches one by one, or globbing, describing a mapping pattern.

For example, the default mapping for the `origin` repository is:

```
[remote "origin"]
  fetch = +refs/heads/*:refs/remotes/origin/*
```

This says that, for example, the contents of the `master` branch (whose full name is `refs/heads/master`) in the remote repository `origin` is to be stored in the local clone of repository in the remote-tracking branch `origin/master` (whose full name is `refs/remotes/origin/master`). The plus `+` sign at the beginning of the pattern tells Git to accept the updates to the remote-tracking branch that are not fast-forward, that is, are not descendants of the previous value.

The mapping can be given using the `fetch` lines in the configuration for the remote, as above, or can be also passed as arguments to a command (it is often enough to specify just the short name of the reference instead

of the full refspec). The configuration is taken into account only if there are no refspecs on the command line.

Fetching and pulling versus pushing

Sending changes (publishing) to the remote repository is done with `git push`, while getting changes from it is done with `git fetch`. These commands send changes in the opposite direction. You should remember, however, that your local repository has the very important difference—it has you sitting at the keyboard available to run other Git commands.

That is why there is no equivalent in the local-to-remote direction to `git pull`, which combines getting and integrating changes (see the next section). There is simply nobody there to resolve possible conflicts (problems doing automated integration).

In particular, there is a difference between how branches and tags are fetched, and how they are pushed. This will be explained in detail later on.

Pull – fetch and update current branch

Many times, you want to incorporate changes from a specific branch of a remote repository into the current branch. The `pull` command downloads changes (running `git fetch` with parameters given); then, it automatically integrates the retrieved branch head into the current branch. By default, it calls `git merge` to integrate changes, but you can make it to run `git rebase` instead. The latter can be done either with the `--rebase` option, or the `pull.rebase` configuration option to `git pull`, or with `branch.<branch name>.rebase` to configure this for the individual branch.

Note that if there is no configuration for the remote (you are doing the pull by URL), Git uses the `FETCH_HEAD` ref to store tips of the fetched branches.

There is also the `git request-pull` command to create information

about published or pending changes for the pull-based workflows, for example, for a variant of the blessed repository workflow. It creates a plain text equivalent of the GitHub merge requests, one which is particularly suitable to send by e-mail.

Pushing to the current branch in a nonbare remote repository

Usually, the repositories you push to are created for synchronization and are bare, that is, without a working area. A bare repository doesn't even have the concept of the current branch (`HEAD`)—there is no work tree, therefore, there is no checked out branch.

Sometimes, however, you might want to push to the nonbare repository. This may happen, for example, as a way of synchronizing two repositories, or as a mechanism for deployment (for example, of a web page or a web application). By default, Git on the server (in the nonbare repository you are pushing into) will deny the ref update to the currently checked out branch. This is because it brings `HEAD` out of sync with the working tree and the staging area, which is very confusing if you don't expect it. You can, however, enable such a push by setting `receive.denyCurrentBranch` to `warn` or `ignore` (changing it from the default value of `refuse`).

You can even make Git update the working directory (which must be clean, that is, without any uncommitted changes) by setting the said configuration variable to `updateInstead`.

An alternative and a more flexible solution to using `git push` for deployment is to configure appropriate hooks on the receiving side—see [Chapter 10, Customizing and Extending Git](#), for information on hooks in general, and [Chapter 11, Git Administration](#), for details on their use on the server.

The default fetch refs and push modes

We usually fetch from public repositories with all the branches made public. We most often want to get a full update of all the branches.

That's why `git clone` sets up the default fetch refspec in a way shown in the *Refspec – remote to local branch mapping specification* section of this chapter. The common exception to "fetch all" rule is following a pull request. But in this case, we have the repository and the branch (or the signed tag) stated explicitly in the request, and we will run the pull command with provided parameters: `git pull <URL> <branch>`.

On the other side, in the private working repository, there are usually many branches that we don't want to publish or, at least, we don't want to publish them yet. In most cases, we would want to publish a single branch: the one we were working on and the one we know is ready. However, if you are the integration manager, you would want to publish a carefully selected subset of the branches instead of just one single branch.

This is yet another difference between fetching and pushing. That's why Git doesn't set up push refspec by default (you can configure it manually nonetheless), but instead relies on the so-called *push modes* (configured using `push.default`) to decide what should be pushed where. This configuration variable, of course, applies only while running the `git push` command without branches to push stated explicitly on the command line.

Tip

Using `git push` to sync out of a host that one cannot pull from

When you work on two machines, `machineA` and `machineB`, each with its own work tree, a typical way to synchronize between them is to run `git pull` from each other. However, in certain situations, you may be able to make the connection only in one direction, but not in the other (for example, because of a firewall or intermittent connectivity). Let's assume that you can fetch and push from `machineB`, but you cannot fetch from `machineA`.

You want to perform push from `machineB` to `machineA` in such way, that the result of the operation is practically indistinguishable from doing

fetch while being on `machineA`. For this you need to specify, via `refs`, that you want to push local branch into its remote-tracking branch.

```
machineB$ git push machineA:repo.git \
  refs/heads/master:refs/remotes/machineB/master
```

The first parameter is the URL in the scp-like syntax, the second parameter is `refs`. Note that you can set these all up in the `config` file in case you need to do something like this more often.

Fetching and pushing branches and tags

The next section will describe which push modes are available and when to use them (for which collaboration workflows). But first, we need to know how Git behaves with respect to tags and branches while interacting with remote repositories.

Because, pushing is not the exact opposite of fetching, and because branches and tags have different objectives (branches point to the lines of development and tags name specific revisions), their behavior is subtly different.

Fetching branches

Fetching branches is quite simple. With the default configuration, the `git fetch` command downloads changes and updates remote-tracking branches (if possible). The latter is done according to the fetch `refs` for the remote.

There are, of course, exceptions to this rule. One such exception is mirroring the repository. In this case all the refs from the remote repository are stored under the same name in the local repository. The `git clone --mirror` would generate the following configuration for `origin`:

```
[remote "origin"]
  url = https://git.example.com/project
  fetch = +refs/*:refs/*
  mirror = true
```

The names of refs that are fetched, together with the object names they point at, are written to the `.git/FETCH_HEAD` file. This information is used, for example, by `git pull`; this is necessary if we are fetching via URL and not via a remote name. It is done because, when we fetch by the URL, there are simply no remote-tracking branches to store the information on the fetched branch to be integrated.

You can delete remote-tracking branches on case by case basis with `git branch -r -d`; you can remove on case by case basis remote-tracking branches for which the corresponding branch in the remote repository no longer exists with `git remote prune` (or in modern Git with `git fetch --prune`).

Fetching tags and automatic tag following

The situation with tags is a bit different. While we would want to make it possible for different developers to work independently on the same branch (for example, an integration branch such as `master`), though in different repositories, we would need all developers to have one specific tag to always refer to the same specific revision. That's why the position of branches in remote repositories is stored using a separate per-remote namespace `refs/remotes/<remote name>/*` in remote-tracking branches, but tags are mirrored—each tag is stored with the same name, in `refs/tags/*` namespace.

Note

Though where the positions of tags in the remote repository are stored can, of course, be configured with the appropriate fetch refspec; Git is that flexible. One example where it might be necessary is the fetching of a subproject, where we want to store its tags in a separate namespace (more information on this issue in [Chapter 9, Managing Subprojects - Building a Living Framework](#)).

This is also why, by default, while downloading changes, Git would also fetch and store locally all the tags that point to the downloaded objects. You can disable this automatic tag following with the `--no-tags` option. This option can be set on the command line as a parameter, or it can be

configured with the `remote.<remote name>.tagopt` setting.

You can also make Git download all the tags with the `--tags` option, or by adding the appropriate fetch refspec value for tags:

```
fetch = +refs/tags/*:refs/tags/*
```

Pushing branches and tags

Pushing is different. Pushing branches are (usually) governed by the selected push mode. You push a local branch (usually just a single current branch) to update a specific branch in the remote repository, from `refs/heads/` locally to `refs/heads/` in remote. It is usually a branch with the same name, but it might be a differently named branch configured as upstream—details will be provided later. You don't need to specify the full refspec: using the ref name (for example, name of a branch) means pushing to the ref with the same name in the remote repository, creating it if it does not exist. Pushing `HEAD` means pushing the current branch into the branch with the same name (not to the `HEAD` in remote—it usually does not exist).

Usually, you push tags explicitly with `git push <remote repository> <tag>` (or `tag <tag>` if by accident there is both a tag and branch with the same name—both mean the `+refs/tags/<tag>:refs/tags/<tag>` refspec). You can push all the tags with `--tags` (and with appropriate refspec), and turn on the automatic tag following with `--follow-tags` (it is not turned on by default as it is for fetch).

As a special case of refspec, pushing an "empty" source into some ref in remote deletes it. The `--delete` option to `git push` is just a shortcut for using this type of refspec. For example, to delete a ref matching `experimental` in the remote repository, you can run:

```
$ git push origin :experimental
```

Note that the remote server might forbid the deletion of refs with `receive.denyDeletes` or hooks.

Push modes and their use

The behavior of `git push`, in the absence of the parameters specifying what to push, and in the absence of the configured push refspec, is specified by the push mode. Different modes are available, each suitable for different collaborative workflows from [Chapter 5, Collaborative Development with Git](#).

The simple push mode – the default

The default push mode in Git 2.0 and later is the so-called `simple` mode. It was designed with the idea of *minimum surprise*: the idea that it is better to prevent publishing a branch, than to make some private changes accidentally public.

With this mode, you always push the current local branch into the same named branch in the remote repository. If you push into the same repository you fetch from (the centralized workflow), it requires the upstream to be set for the current branch. The upstream is named the same as the branch.

This means that, in the centralized workflow (push into the same repository you fetch from), it works like `upstream` with the additional safety that the upstream must have the same name as the current (pushed) branch. With triangular workflow, while pushing to a remote that is different from the remote you normally pull from, it works like `current`.

This is the safest option; it is well-suited for beginners, which is why it is the default mode. You can turn it on explicitly with `git config push.default simple`.

The matching push mode for maintainers

Before version 2.0 of Git, the default push mode was `matching`. This mode is most useful for the maintainer (also known as the integration manager) in a blessed repository workflow. But most of the Git users are not maintainers; that's why the default push mode was changed to `simple`.

The maintainer would get contributions from other developers, be it via pull request or patches sent in an e-mail, and put them into topic branches. He or she could also create topic branches for their own contributions. Then, the topic branches considered to be suitable are merged into the appropriate integration branches (for example, `maint`, `master`, and `next`) – merging will be covered in [Chapter 7, Merging Changes Together](#). All this is done in the maintainer's private repository.

The public blessed repository (one that everyone fetches from, as described in [Chapter 5, Collaborative Development with Git](#)) should contain only long-running branches (otherwise, other developers could start basing their work on a branch that suddenly vanishes). Git cannot know by itself which branches are long-lived and which are short-lived.

With the matching mode, Git will push all the local branches that have their equivalent with the same name in the remote repository. This means that only the branches that are already published will be pushed to the remote repository. To make a new branch public you need to push it explicitly the first time, for example:

```
$ git push origin maint-1.4
```

Note

Note that with this mode, unlike with other modes, using `git push` command without providing list of branches to push can publish multiple branches at once, and may not publish the current branch.

To turn on the matching mode globally, you can run:

```
$ git config push.default matching
```

If you want to turn it on for a specific repository, you need to use a special refspec composed of a sole colon. Assuming that the said repository is named `origin` and that we want a not forced push, it can be done with:

```
$ git config remote.origin.push :
```

You can, of course, push matching branches using this refspec on the command line:

```
$ git push origin :
```

The upstream push mode for the centralized workflow

In the centralized workflow, there is the single shared central repository every developer with commit access pushes to. This shared repository will have only long-lived integration branches, usually only `maint` and `master`, and sometimes only `master`.

One should rather never work directly on `master` (perhaps with the exception of simple single-commit topics), but rather fork a topic branch for each separate feature out of the remote-tracking branch:

```
$ git checkout -b feature-foo origin/master
```

In the centralized workflow, the integration is distributed: each developer is responsible for merging changes (in their topic branches), and publishing the result to the `master` branch in the central repository. You would need to update the local `master` branch, merge the topic branch to it, and push it:

```
$ git checkout master
$ git pull
$ git merge feature-foo
$ git push origin master
```

An alternate solution is to rebase the topic branch on the top of the remote-tracking branch, rather than merging it. After rebasing, the topic branch should be an ancestor of `master` in the remote repository, so we can simply push it into `master`:

```
$ git checkout feature-foo
$ git pull --rebase
$ git push origin feature-foo:master
```

In both the cases, you are pushing the local branch (`master` in the merge-based workflow, the feature branch in the rebase-based workflow) into the branch it tracks in the remote repository; in this case,

origin's master.

That is what the `upstream` push mode was created for:

```
$ git config push.default upstream
```

This mode makes Git push the current branch to the specific branch in the remote repository—the branch whose changes are usually integrated into the current branch. This branch in the remote repository is the upstream branch (and can be referenced as `@{upstream}`). Turning this mode on makes it possible to simplify the last command in both examples to the following:

```
$ git push
```

The information about the upstream is created either automatically (while forking off the remote-tracking branch), or explicitly with the `--track` option. It is stored in the configuration file and it can be edited with ordinary configuration tools. Alternatively, it can be changed later with the following:

```
$ git branch --set-upstream-to=<branchname>
```

The current push mode for the blessed repository workflow

In the blessed repository workflow, each developer has his or her own private and public repository. In this model, one fetches from the blessed repository and pushes to his or her own public repository.

In this workflow, you start working on a feature by creating a new topic branch for it:

```
$ git checkout -b fix-tty-bug origin/master
```

When the features are ready, you push it into your public repository, perhaps rebasing it first to make it easier for the maintainer to merge it:

```
$ git push origin fix-tty-bug
```

Here, it is assumed that you used `pushurl` to configure the triangular workflow, and the push remote is `origin`. You would need to replace `origin` here with the appropriate name of the publishing remote if you are using a separate remote for your own public repository (using a separate repository makes it possible to use it not only for publishing, but also for synchronization between different machines).

To configure Git so when on `fix-tty-bug` branch it is enough to just run `git push`, you need to set up Git to use the `current` push mode, which can be done with the following:

```
$ git config push.default current
```

This mode will push the current branch to the branch with the same name at the receiving end.

Note that, if you are using a separate remote for the publishing repository, you would need to set up the `remote.pushDefault` configuration option to be able to use just `git push` for publishing.

Summary

This chapter has shown how to effectively use branches for development and for collaboration.

We got to know a wide set of the various uses of branches, from integration, through release management and the parallel development of features, to fixing bugs. You have learned different branching workflows, including the very useful and widely used topic branch workflow. The knowledge should help you make the best use of branching, customizing the model of work to fit the project and your own preferences.

You have also learned how to deal with multiple branches per repository while downloading or publishing changes. Git provides flexibility in how the information on branches and other refs in the remote repository are managed using the so-called `refsspecs` to define mapping to local refs: remote-tracking branches, local branches, and tags. Usually, fetching is governed by `fetch refspec`, but pushing is managed by the configured push mode. Various collaborative workflows require a different handling of branch publishing; this chapter describes which push mode to use with which workflow and explains why.

You also got to know a few useful tricks. One of them is how to store the project's web page or the generated HTML documentation in a single repository with the "orphan" branch trick (which is used, for example, by GitHub Project Pages). You found out how to synchronize the working directory of the remote repository (for example, for the deployment of a web application) with `git push`; one of the possible solutions. You have learned how to do fetch equivalent with push, if the connection is possible only in the opposite direction.

The next chapter, [Chapter 7, Merging Changes Together](#), will explain how to integrate changes from other branches and other developers. You will learn about merging and rebasing, and how to deal with situations where Git could not do it automatically (how to handle various types of

merge conflicts). You will also learn about cherry-picking and reverting commits.

Chapter 7. Merging Changes Together

The previous chapter, *Advanced Branching Techniques*, described how to use branches effectively for collaboration and development.

This chapter will teach you how to integrate changes from different parallel lines of development (that is, branches) together by creating a merge commit, or by reapplying changes with the rebase operation. Here, the concepts of merge and rebase are explained, including the differences between them and how they both can be used. This chapter will also explain the different types of merge conflicts, and teach how to examine them, and how to resolve them.

In this chapter, we will cover the following topics:

- Merging, merge strategies, and merge drivers
- Cherry-picking and reverting a commit
- Applying a patch and a patch series
- Rebasing a branch and replaying its commits
- Merge algorithm on file and contents level
- Three stages in the index
- Merge conflicts, how to examine them, and how to resolve them
- Reusing recorded [conflict] resolutions with `git rerere`
- External tool: `git-imerge`

Methods of combining changes

Now that you have changes from other people in the remote-tracking branches (or in the series of e-mails), you need to combine them, perhaps also with your changes. Or perhaps, your work on a new feature, created and performed on a separate topic branch, is now ready to be included in the long-lived development branch, and made available to other people. Maybe you have created a bugfix and want to include it in all the long-lived graduation branches. In short, you want to join two

divergent lines of development, to combine them together.

Git provides a few different methods of combining changes and variations of these methods. One of these methods is a *merge* operation, joining two lines of development with a two-parent commit. Another way to copy introduced work from one branch to another is via cherry-picking, which is creating a new commit with the same changeset on another line of development (this is sometimes necessary to use). Or, you can reapply changes, transplanting one branch on top of another with *rebase*. We will now examine all these methods and their variants, see how they work, and when they can be used.

In many cases, Git will be able to combine changes automatically; the next section will talk about what you can do if it fails and if there are merge conflicts.

Merging branches

The merge operation joins two (or more) separate branches together, including *all* the changes since the point of divergence into the current branch. You do this with the `git merge` command:

```
$ git checkout master
$ git merge bugfix123
```

Here, we first switched to a branch we want to merge into (`master` in this example), and then provided the branch to be merged (here, `bugfix123`).

No divergence – fast-forward and up-to-date cases

Say that you need to create a fix for a bug somebody found. Let's assume that you have followed the recommendations of the topic branch workflow from [Chapter 6, Advanced Branching Techniques](#), and created a separate bugfix branch, named `bugfix123`, off the maintenance branch `maint`. You have run your tests (that were perhaps just created), making sure that the fix is correct and is what you want. Now you are ready to merge it, at least, into `maint` to make this fix

available for other people, and perhaps, also into `master` (into the stable branch). The latter can be configured to deploy the fix to production environment.

In such cases, there is often no real divergence, which means that there were no commits on the maintenance branch (the branch we are merging into), since a bugfix branch was created. Because of this, Git would, by default, simply move the branch pointer of the current branch forward:

```
$ git checkout maint
$ git merge i18n
Updating f41c546..3a0b90c
Fast-forward
 src/random.c | 2 ++
 1 file changed, 2 insertions(+)
```

You have probably seen this `Fast-forward` phrase among output messages during `git pull`, when there are no changes on the branch you are pulling into. The fast-forward merge situation is shown on Fig. 1:

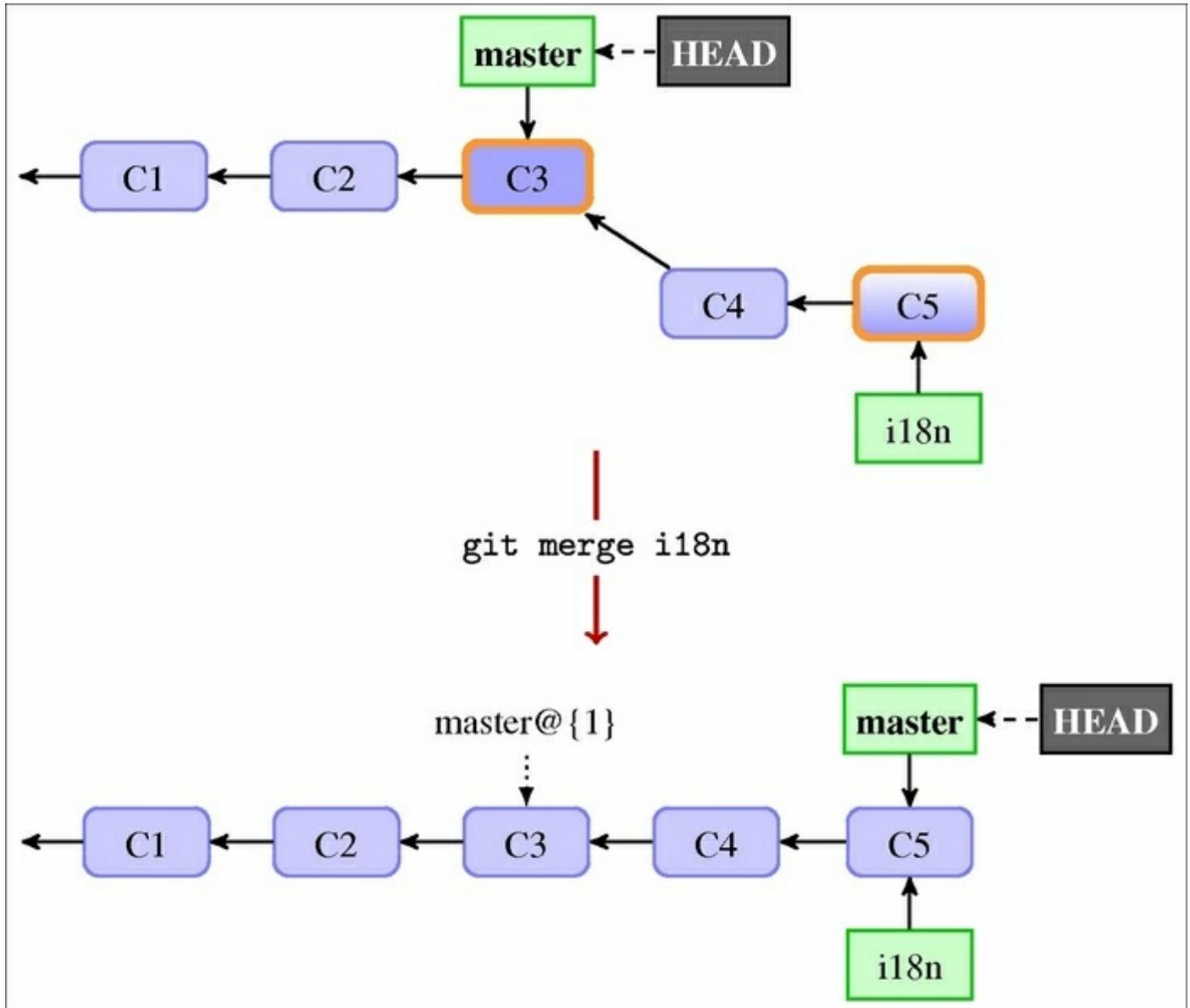


Fig 1: The `master` branch is fast-forwarded to `i18n` during merge

This case is important for the centralized and the peer-to-peer workflows (described in [Chapter 5, Collaborative Development with Git](#)), as it is the fast-forward merge that allows you to ultimately push your changes forward.

In some cases, it is not what you want. See that, for example, after the fast-forward merge in *Fig 1*, we have lost the information that the **C4** and **C5** commits were done on the `i18n` topic branch, and are a part of internationalization efforts. We can force creating a merge commit (described in the next section) even in such cases with the `git merge --`

`no-ff` command. The default is `--ff`; to fail instead of creating a merge commit you can use `--ff-only` (ensuring fast-forward only).

There is another situation where the head (tip) of one branch is the ancestor of the other, namely, the up-to-date case where the branch we are trying to merge is already included (merged) in the current branch. Git doesn't need to do anything in this case; it just informs the user about it.

Creating a merge commit

When you are merging fully fledged feature branches, rather than merging bugfix branches as in the previous section, the situation is usually different from the previously described `Fast-forward case`. Then, the development usually had diverged. You began work on a feature of a topic branch to separate and isolate it from other developments.

Suppose that you have decided that your work on a feature (for example, work on adding support for internationalization on the `i18n` topic branch) is complete and ready to be included in the `master` stable branch. In order to do so with a merge operation, you need to first check out the branch you want to merge into, and then run the `git merge` command with the branch being merged as a parameter:

```
$ git checkout master
Switched to branch 'master'
$ git merge i18n
Merge made by the 'recursive' strategy.
Src/random.c |    2 ++
1 file changed, 2 insertions(+)
```

Because the top commit on the branch you are on (and are merging into) is not a direct ancestor or a direct descendant of the branch you are merging in, Git has to do more work than just moving the branch pointer. In this case, Git does a merge of changes since the divergence, and stores it as a merge commit on the current branch. This commit has two parents denoting that it was created based on more than one commit (more than one branch): the first parent is the previous tip of the current

branch and the second parent is the tip of branch you are merging in.

Note

Note that Git does commit the result of merge if it can be done automatically (there are no conflicts). But the fact that the merge succeeded at the text level doesn't necessarily mean that the merge result is correct. You can either ask Git to not autocommit a merge with `git merge --no-commit` to examine it first, or you can examine the merge commit and then use the `git commit --amend` command if it is incorrect.

In contrast, most other version control systems do not automatically commit the result of a merge.

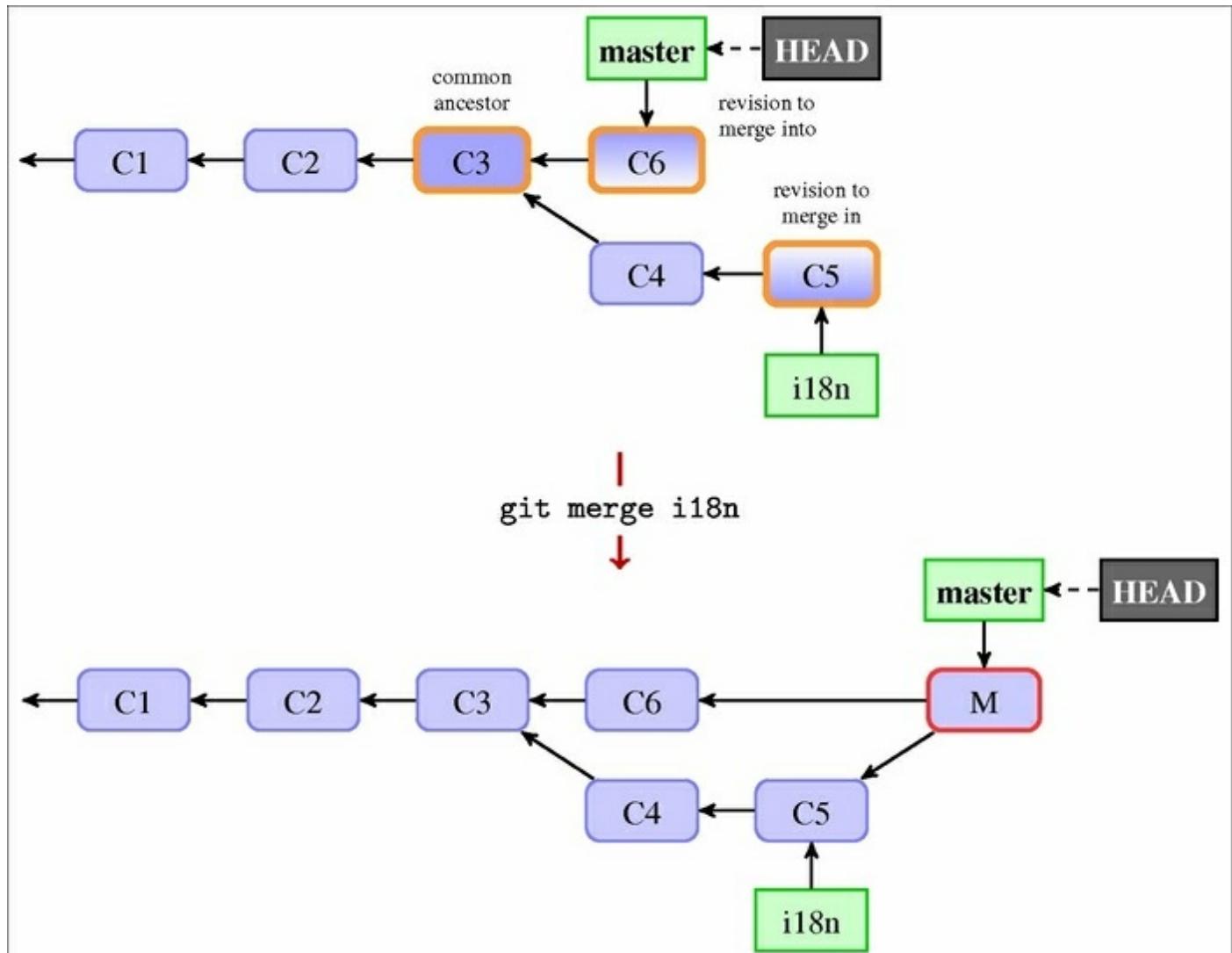


Fig 2: Three revisions used in a typical merge and the resulting merge commit

Git creates contents of a merge commit (**M** in Fig 2) using by default (and in most cases) the three way merge, which in turn uses the snapshots pointed to the tips of the branches being merged (`master`: **C6** and `i18n`: **C5**) and the common ancestor of the two (**C3** here, which you can find with the `git merge-base` command).

It's worth pointing out that Git can determine the common ancestor automatically thanks to storing revisions in the DAG and remembering merges. This was not the case in the older revision control systems.

A very important issue is that Git creates the merge commit contents based usually only on the three revisions: merged into (ours), merged in (theirs), and the common ancestor (merge base). It does not examine what had happened on the divergent parts of the branches; this is what makes merging fast. But because of this, Git also does not know about the cherry-picked or reverted changes on the branches being merged, which might lead to surprising results (see, for example, the section about reverting merges in [Chapter 8, Keeping History Clean](#)).

Merge strategies and their options

In the merge message, we have seen that it was made by the *recursive* strategy. The *merge* strategy is an algorithm that Git uses to compose the result of joining two or more lines of development, which is basing this result on the DAG of revisions.

There are a few merge strategies that you can select to use with the `--strategy`/ `-s` option. By default, Git uses the recursive merge strategy while joining two branches and a very simple *octopus* merge strategy while joining more than two branches. You can also choose the *resolve* merge strategy if the default one fails; it is fast and safe, though less capable in merging.

The two remaining merge strategies are special purpose algorithms. The *ours* merge strategy can be used when we want to abandon changes in the merged in branch, but keep them in the history of the merged into

branch, for example, for documentation purposes. This strategy simply repeats the current snapshot (ours version) as a merge commit. Note that ours merge strategy, invoked with `--strategy=ours` or `-s ours`, should be not confused with the "ours" option to the default recursive merge strategy, `--strategy=recursive --strategy-option=ours` or just `-xours`, which means something different.

The *subtree* merge strategy can be used for subsequent merges from an independent project into a subdirectory (subtree) in a main project. It automatically figures out where the subproject was put. This issue, and the idea of subtrees, will be described in more detail in [Chapter 9, Managing Subprojects – Building a Living Framework](#).

The default *recursive* merge strategy is named after how it deals with multiple merge bases and criss-cross merges. In case of more than one merge base (more than one common ancestor that can be used for a three-way merge), it creates a merge tree (conflicts and all) from the ancestors as a merge base, that is, it merges recursively. Of course, these common ancestors being merged can have more than one merge base again.

Some strategies are customizable and take their own options. You can pass an option to a merge algorithm with `-x<option>` (or `--strategy-option=<option>`) on the command line, or set it with the appropriate configuration variables. You will find more about merge options in a later section, when we will be talking about solving merge conflicts.

Reminder – merge drivers

[Chapter 4, Managing Your Worktree](#), introduced `gitattributes`, among others merge drivers. These *drivers* are user-defined and deal with merging file contents if there is a conflict, replacing the default three-way file-level merge. Merge strategies in contrast deal with the DAG level merging (and tree-level, that is, merging directories) and you can only choose from the built-in options.

Reminder – signing merges and merging tags

In [Chapter 5, Collaborative Development with Git](#), you have learned about signing your work. While using merge to join two lines of development, you can either merge a signed tag or sign a merge commit (or both). Signing a merge commit is done with the `-s` / `--gpg-sign` option to use the `git merge` or the `git commit` command; the latter is used if there were conflicts or the `--no-commit` option was used while merging.

Copying and applying a changeset

The merging operation is about joining two lines of development (two branches), including all the changes since their divergence. This means, as described in [Chapter 6, Advanced Branching Techniques](#), that if there is one commit on the less stable branch (for example, `master`) that you want to have also in a more stable branch (for example, `maint`), you cannot use the merge operation. You need to create a copy of such commit. Entering such situation should be avoided (using topic branches), but it can happen, and handling it is sometimes necessary.

Sometimes, the changes to be applied come not from the repository (as a revision in the DAG to be copied), but in the form of a patch, that is, a unified diff or an e-mail generated with `git format-patch` (with `patch`, plus a commit message). Git includes the `git am` tool to handle mass applying of commit-containing patches.

Both of these are useful on their own, but understanding these methods of getting changes is necessary to understand how rebasing works.

Cherry-pick – creating a copy of a changeset

You can create a copy of a commit (or a series of commits) with the `cherry-pick` command. Given a series of commits (usually, just a single commit), it applies the changes each one introduces, recording a new commit for each.

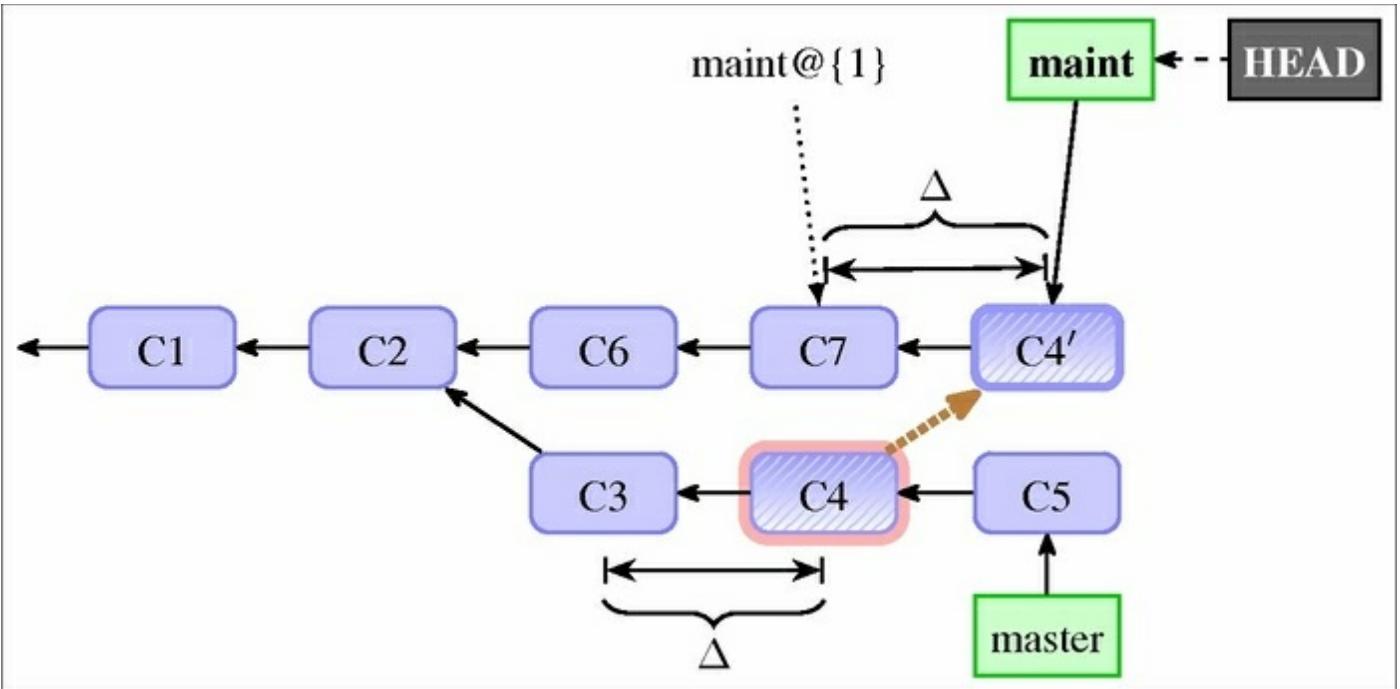


Fig 3: Cherry-picking a commit from `master` to `maint`. The thick brown dotted line from `c4` to `c4'` denotes copy; it is not a reference.

This does not mean that the snapshot (that is, the state of a project) is the same in the original and in the copy; the latter will include other changes. Also, while the changes will usually be the same (as in Fig 3), they can also in some cases be different, for example if part of the changes was already present in the earlier commits.

Note that, by default, Git does not save information about where the cherry-picked commit came from. You can append this information to an original commit message, as a (`cherry-picked from the commit <sha-1>`) line with `git cherry-pick -x <commit>`. This is only done for cherry-picks without conflicts. Remember that this information is only useful if you have an access to the copied commit. Do not use it if you are copying commits from the private branch, as other developers won't be able to make use of that information.

Revert – undoing an effect of a commit

Sometimes it turns out that, even with code review, there will be some bad commits that you need to back out (perhaps it turned out to be a not

so good idea, or it contains bugs). If the commit is already made public, you cannot simply remove it. You need to undo its effects; this issue will be explained in detail in [Chapter 8, Keeping History Clean](#).

This "undoing of a commit" can be done by creating a commit with a reversal of changes, something like cherry-pick but applying the reverse of changes. This is done with the `revert` command.

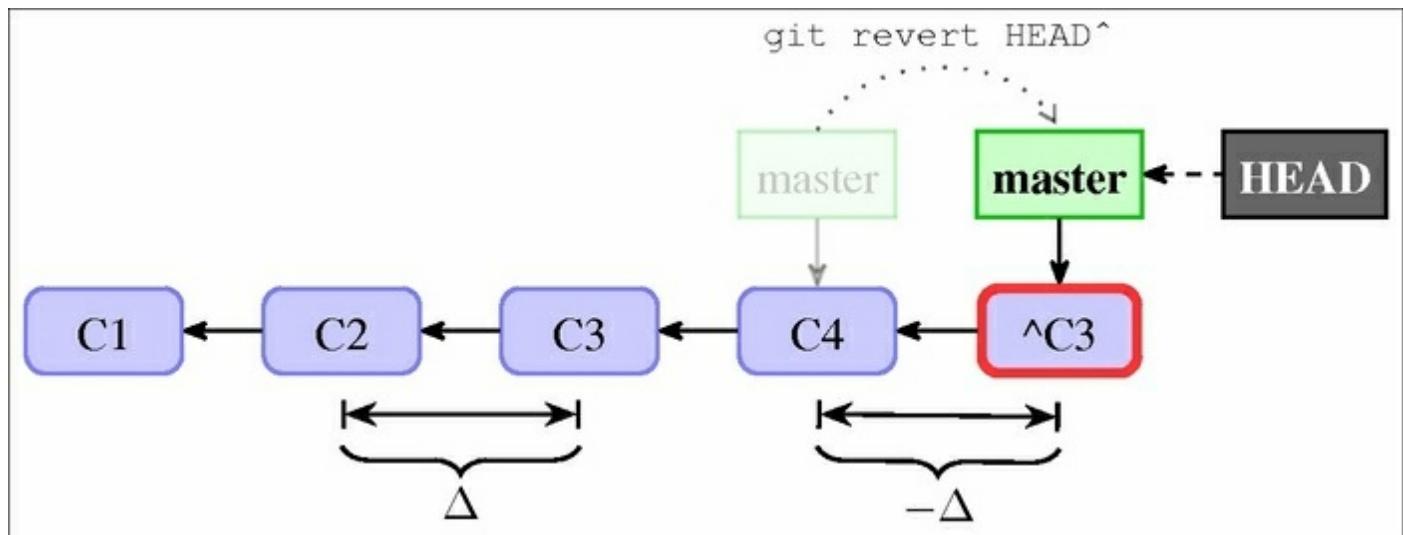


Fig 4: The effect of using `git revert C3` on a master branch, creating a new commit named `^C3`

The name of this operation might be misleading. If you want to revert all the changes made to the whole working area, you can use `git reset` (in particular, the `--hard` option). If you want to revert changes made to a single file, use `git checkout <file>`. Both of these are explained in detail in [Chapter 4, Managing Your Worktree](#). The `git revert` command records a new commit to reverse the effect of the earlier commit (often, a faulty one).

Applying a series of commits from patches

Some collaborative workflows include exchanging the changes as patches via an e-mail (or another communication medium). This workflow is often encountered in open-source projects; it is often easier for a new or a sporadic contributor to create a specially crafted e-mail

(for example, with `git format-patch`) and send it to a maintainer or a mailing list, than to set up a public repository and send a pull request.

You can apply a series of patches from a mailbox (in the mbox or maildir format; the latter is just a series of files) with the `git am` command. If these emails (or files) were created from the `git format-patch` output, you can use `git am --3way` to use the three-way file merge in the case of conflicts. Resolving conflicts will be discussed in later section of this chapter.

Note

You can find both tools to help use the patch submission process by sending a series of patches, for example from the `pull` request on GitHub (for example, the `submitGit` web app for Git project), and tools that track web page patches sent to a mailing list (for example, the `patchwork` tool).

Cherry-picking and reverting a merge

This is all good, but what happens if you want to cherry-pick or revert a merge commit? Such commits have more than one parent, thus they have more than one change associated with them.

In this case, you have to tell Git which change you want to pick up (in the case of cherry-pick), or back out (in the case of revert) with the `-m <parent number>` option.

Note that reverting a merge undoes the changes, but it does not remove the merge from the history of the project. See the section on reverting merges in [Chapter 8, Keeping History Clean](#).

Rebasing a branch

Besides merging, Git supports additional way to integrate changes from one branch into another: namely the rebase operation.

Like a merge, it deals with the changes since the point of divergence (at

least, by default). But while a merge creates a new commit by joining two branches, rebase takes the new commits from one branch (takes the commits since the divergence) and reapplies them on top of the other branch.

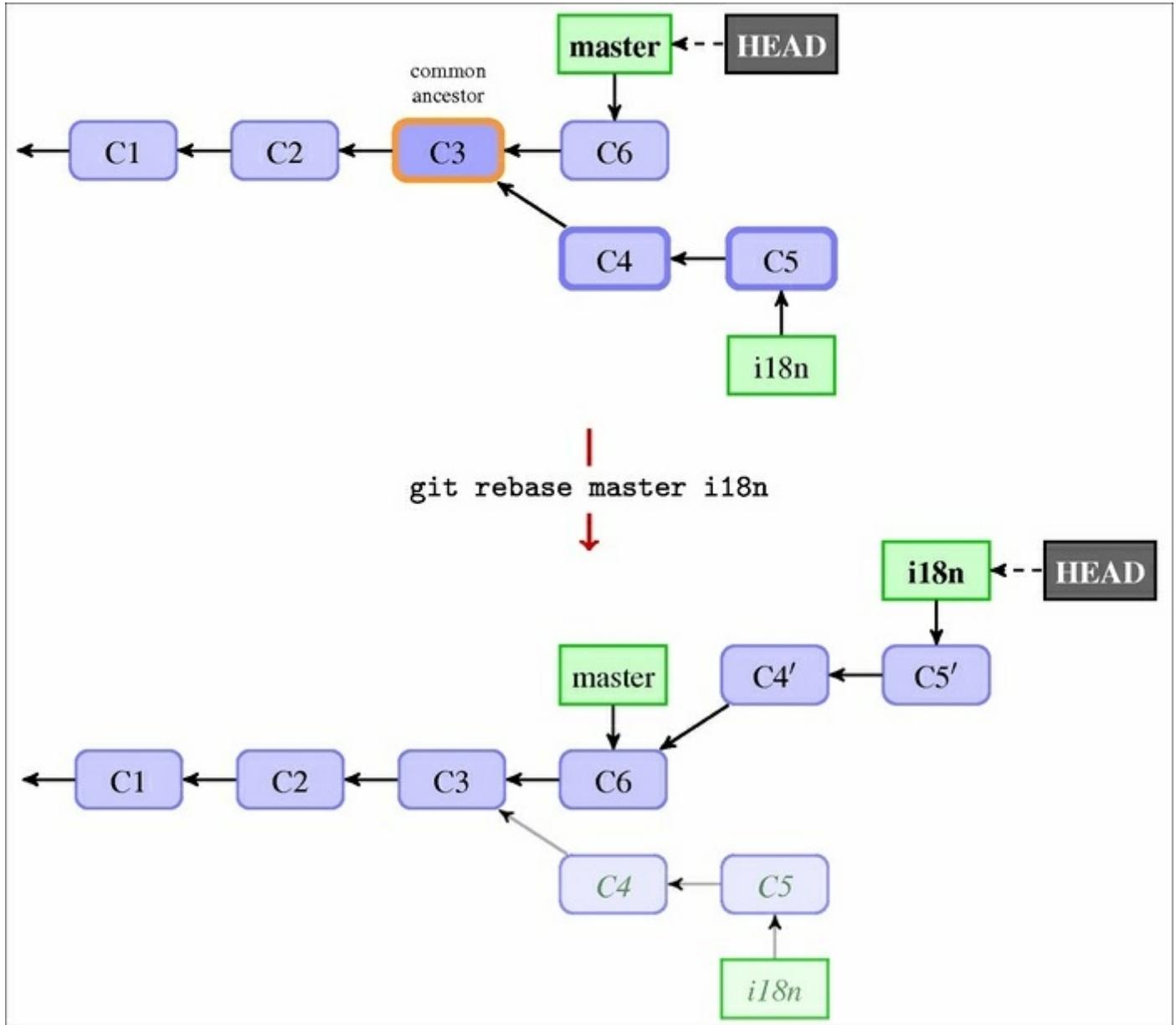


Fig 5: Effects of the rebase operation

With merge, you first switched to the branch to be merged and then used the `merge` command to select a branch to merge in. With rebase, it is a bit different. First you select a branch to rebase (changes to reapply) and then use the `rebase` command to select where to put it. In both the cases, you first check out the branch to be modified, where a new

commit or commits would be (a merge commit in the case of merging, and a replay of commits in the case of rebasing):

```
$ git checkout i18n
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: Mark messages for translation
```

Or, you can use `git rebase master i18n` as a shortcut. In this form, you can easily see that the rebase operation takes the `master..i18n` range of revisions (this notation is explained in [Chapter 2, Exploring Project History](#)), replays it on top of `master`, and finally points `i18n` to the replayed commits.

Note that old versions of commits doesn't vanish, at least not immediately. They would be accessible via `reflog` (and `ORIG_HEAD`) for a grace period. This means that it is not that hard to check how replaying changed the snapshots of a project, and with a bit more effort how changesets themselves have changed.

Merge versus rebase

We have these two ways of integrating changes: merge and rebase. How do they differ and what are their advantages and disadvantages? You can compare *Fig 2* in the *Creating a merge commit* section with *Fig 5* in the *Rebasing a branch* section.

First, merge doesn't change history (see [Chapter 8, Keeping History Clean](#)). It creates and adds a new commit (unless it was a fast-forward merge; then it just advances the branch head), but the commits that were reachable from the branch remain reachable. This is not the case with rebase. Commits get rewritten, old versions are forgotten, and the DAG of revisions changes. What was once reachable might no longer be reachable. This means that you should not rebase published branches.

Secondly, merge is a one-step operation with one place to resolve merge conflicts. The rebase operation is multi-step; the steps are smaller (to keep changes small, see [Chapter 12, Git Best Practices](#)), but there are more of them.

Linked to this is a fact that the merge result is based (usually) on three commits only, and that it does not take into the account what happened on either of the branches being integrated step by step; only the endpoints matter. On the other hand, rebase reapplies each commit individually, so the road to the final result matters here.

Thirdly, the history looks different: you get a simple linear history with rebase, while using the merge operation leads to complex history with the lines of development forking and joining. The history is simpler for rebase, but you lose information that the changes were developed on a separate branch and that they were grouped together, which you get with merge (at least with `--no-ff`). There is even the `git-resurrect` script in the `Git contrib` tools, that uses the information stored in the commit messages of the merge commits to resurrect the old, long deleted feature branches.

The last difference is that, because of the underlying mechanism, rebase does not, by default, preserve merge commits while reapplying them. You need to explicitly use the `--preserve-merges` option. The merge operation does not change the history, so merge commits are left as it is.

Types of rebase

The previous section described two mechanisms to copy or apply changes: the `git cherry-pick` command, and the pipeline from `git format-patch` to `git am --3way`. Either of them can be used by `git rebase` to reapply commits.

The default is to use the patch-based workflow, as it is faster. With this type of rebase, you can use some additional options with rebase, which are actually passed down to the `git apply` command that does the actual replaying of changesets. These options will be described later while talking about conflicts.

Alternatively, you can use the `--merge` option to utilize merge strategies to do the rebase (kind of cherry-picking each commit). The default recursive merge strategy allows rebase to be aware of the renames on

the upstream side (where we put the replayed commits). With this option, you can also select a specific merge strategy and pass options to it.

There is also an interactive rebase with its own set of options. This is one of the main tools in [Chapter 8](#), *Keeping History Clean*. It can be used to execute tests after each replayed commit to check that the replay is correct.

Advanced rebasing techniques

You can also have your rebase operation replay on something other than the target branch of the rebase with `--onto <newbase>`, separating selected range of revisions to replay from the new base to replay onto.

Let's assume that you had based your `featureA` topic branch on the unstable development branch named `next`, because it is dependent on some feature that was not yet ready and not yet present in the stable branch (`master`). If the functionality on which `featureA` depends was deemed stable and was merged into `master`, you would want to move this branch from being forked from the `next` to being forked from `master`. Or perhaps, you started the `server` branch from the related `client` branch, but you want to make it more obvious that they are independent.

You can do this with `git rebase --onto master next featureA` in the first case, and `git rebase --onto master server client` in the second one.

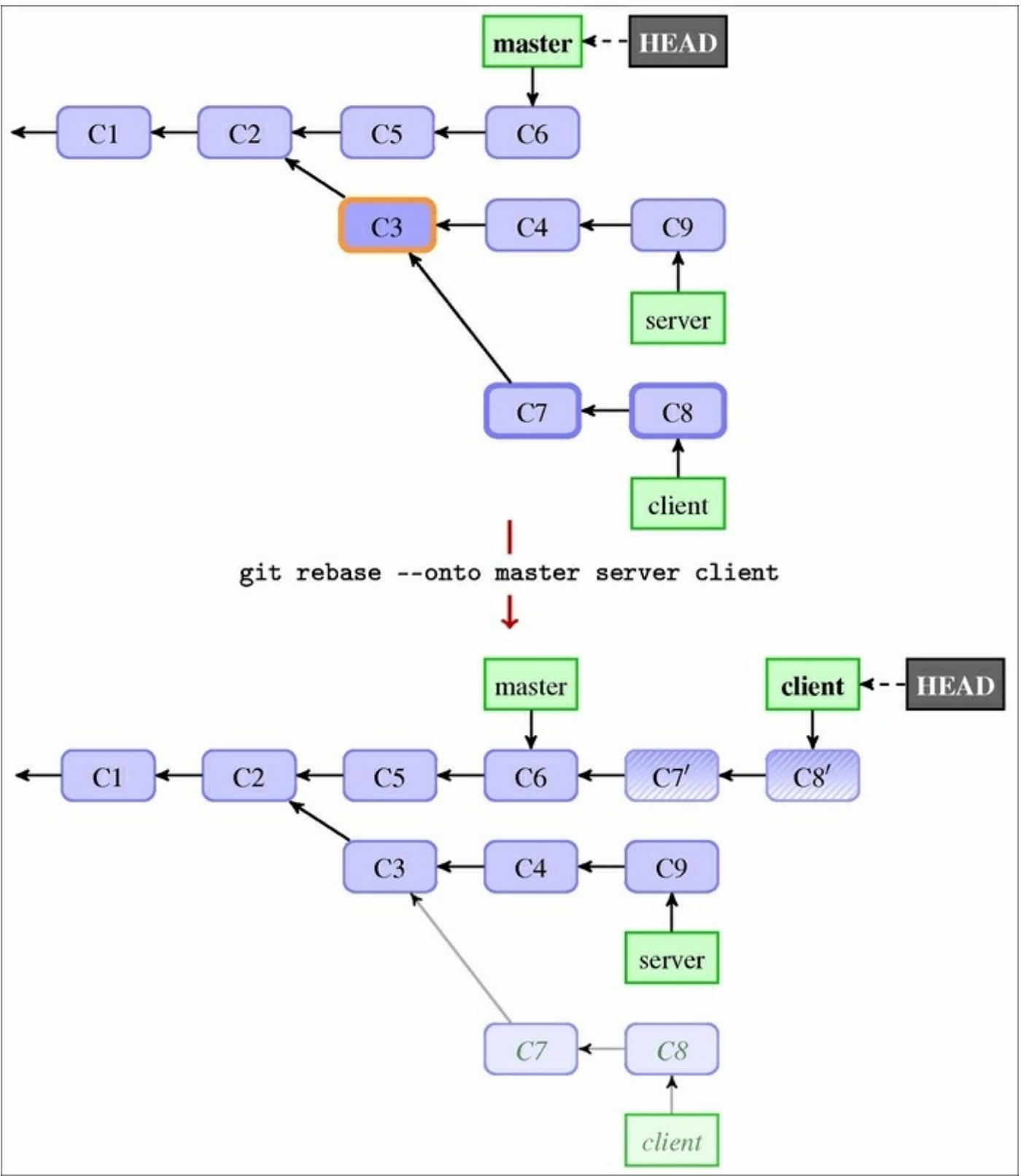


Fig 6: Rebasing branch, moving it from one branch to the other

Or perhaps, you want to rebase only a part of the branch. You can do this with `git rebase --interactive`, but you can also use `git rebase -`

-onto <new base> <starting point> <branch>.

You can even choose to rebase the whole branch (usually, an orphan branch) with the --root option. In this case, you would replay the whole branch and not just a selected subset of it.

Resolving merge conflicts

Merging in Git is typically fairly easy. Since Git stores and has access to the full graph of revisions, it can automatically find where the branches diverged, and merge only those divergent parts. This works even in the case of repeated merges, so you can keep a very long-lived branch up to date by repeatedly merging into it or by rebasing it on top of new changes.

However, it is not always possible to automatically combine changes. There are problems that Git cannot solve, for example because there were different changes to the same area of a file on different branches: these problems are called **merge conflicts**. Similarly, there can be problems while reapplying changes, though you would still get merge conflicts in case of problems.

The three-way merge

Unlike some other version control systems, Git does not try to be overly clever about merge conflict resolutions, and does not try to solve them all automatically. Git's philosophy is to be smart about determining the cases when a merge can be easily done automatically (for example, taking renames into account), and if automatic resolution is not possible, to not be overly clever about trying to resolve it. It is better to bail out and ask users to resolve merge, perhaps unnecessary with a smart algorithm, than to automatically create an incorrect one.

Git uses the three-way merge algorithm to come up with the result of the merge, comparing the common ancestors (*base*), side merged in (*theirs*), and side merged into (*ours*). This algorithm is very simple, at least at the tree level, that is, the granularity level of files. The following table explains the rules of the algorithm:

ancestor (base)	HEAD (ours)	branch (theirs)	result
A	A	A	A

A	A	B	B
A	B	A	B
A	B	B	B
A	B	C	merge

The rules for the trivial tree-level three-way merges are (see the preceding table):

- If only one side changes a file, take the changed version
- If both the sides have the same changes, take the changed version
- If one side has a different change from the other, there is **merge conflict** at the contents level

It is a bit more complicated if there are more than one ancestor or if a file is not present in all the versions. But usually it is enough to know and understand these rules.

If one side changed the file differently from the other (where the type of the change counts, for example, renaming a file on one branch doesn't conflict with the changing contents of the file on the other branch), Git tries to merge the files at the contents level, using the provided merge driver if it is defined, and the contents level three-way merge otherwise (for text files).

The three-way file merge examines whether the changes touch different parts of the file (different lines are changed, and these changes are well separated by more than diff context sizes away from each other). If these changes are present in different parts of the file, Git resolves the merge automatically (and tells us which files are automerged).

However, if you changed the same part of the same file differently in the two branches you're merging together, Git won't be able to merge them cleanly:

```
$ git merge i18n
Auto-merging src/rand.c
CONFLICT (content): Merge conflict in src/rand.c
Automatic merge failed; fix conflicts and then commit the
result.
```

Examining failed merges

In the case Git is unable to automatically resolve a merge (or if you have passed the `--no-commit` option to the `git merge` command), it would not create a merge commit. It will pause the process, waiting for you to resolve the conflict.

You can then always abort the process of merging with `git merge --abort`, in modern Git. With the older version, you would need to use `git reset` and delete `.git/MERGE_HEAD`.

Conflict markers in the worktree

If you want to see which files are yet unmerged at any point after a merge conflict, you can run `git status`:

```
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")
```

```
Unmerged paths:
  (use "git add <file>..." to mark resolution)
```

```
    both modified:      src/rand.c
```

```
no changes added to commit (use "git add" and/or "git commit -a")
```

Anything that has not been resolved is listed as unmerged. In the case of content conflicts, Git uses standard conflict markers, putting them around the place of conflict with the *ours* and *theirs* version of the conflicted area in question. Your file will contain a section that would look somewhat like the following:

```
<<<<< HEAD:src/rand.c
```

```
fprintf(stderr, "Usage: %s <number> [<count>]\n", argv[0]);  
=====  
fprintf(stderr, _("Usage: %s <number> [<count>]\n"), argv[0]);  
>>>>> i18n:src/rand.c
```

This means that the ours version on the current branch (`HEAD`) in the `src/rand.c` file is there at the top of this block between the `<<<<<` and `=====` markers, while the theirs version on the `i18n` branch being merged (also from `src/rand.c`) is there at the bottom part between the `=====` and `>>>>>` markers.

You need to replace this whole block by the resolution of the merge, either by choosing one side (and deleting the rest) or combining both changes, for example:

```
fprintf(stderr, _("Usage: %s <number> [<count>]\n"), argv[0]);
```

To help you avoid committing unresolved changes by mistake, Git by default checks whether committed changes include something that looks like conflict markers, refusing to create a merge commit without `--no-verify` if it finds them.

If you need to examine a common ancestor version to be able to resolve a conflict, you can switch to `diff3` like conflict markers, which have an additional block:

```
<<<<< HEAD:src/rand.c  
fprintf(stderr, "Usage: %s <number> [<count>]\n", argv[0]);  
|||||||  
fprintf(stderr, "Usage: %s <number> [<count>]\n", argv[0]);  
=====  
fprintf(stderr, _("Usage: %s <number> [<count>]\n"), argv[0]);  
>>>>> i18n:src/rand.c
```

You can replace merge conflict markers individually on a file-per-file basis by rechecking the file again with the following command:

```
$ git checkout --conflict=diff3 src/rand.c
```

If you prefer to use this format all the time, you can set it as the default for future merge conflicts, by setting `merge.conflictStyle` to `diff3`

(from the default of `merge`).

Three stages in the index

But how does Git keep track of which files are merged and which are not? Conflict markers in the working directory files would not be enough. Sometimes, there are legitimate contents that look like commit markers (for example, test files for merge, or files in the AsciiDoc format), and there are more conflict types than `CONFLICT(content)`. How does Git, for example, represent the case where both sides renamed the file but in a different way, or where one side changed the file and the other side removed it?

It turns out that it is another use for the staging area of the commit (a merge commit in this case), which is also known as the index. In the case of conflicts, Git stores all of conflicted files versions in the index under `stages`; each stage has a number associated with it. Stage 1 is the common ancestor (`base`), stage 2 is the merged into version from `HEAD`, that is, the current branch (`ours`), and stage 3 is from `MERGE_HEAD`, the version you're merging in (`theirs`).

You can see these stages for the unmerged files with the low level (plumbing) command `git ls-files --unmerged` (or for all the files with `git ls-files --stage`):

```
$ git ls-files --unmerged
100755 ac51efdc3df4f4fd318d1a02ad05331d8e2c9111 1    src/rand.c
100755 36c06c8752c78d2aaaf89571132f3bf7841a7b5c3 2    src/rand.c
100755 e85207e04dfdd50b0a1e9febbc67fd837c44a1cd 3    src/rand.c
```

You can refer to each version with the `:<stage number>:<pathname>` specifier. For example, if you want to view a common ancestor version of `src/rand.c`, you can use the following:

```
$ git show :1:src/rand.c
```

If there is no conflict, the file is in stage 0 of the index.

Examining differences – the combined diff format

You can use the `status` command to find which files are unmerged, and conflict markers do a good job of showing conflicts. How to see only conflicts before we work on them, and how to see how they were resolved? The answer is `git diff`.

One thing to remember is that for merges, even the merges in progress, Git will show the so-called **combined diff** format. It will look as follows (for a conflicted file during a merge):

```
$ git diff
diff --cc src/rand.c
index 293c8fc,4b87d29..0000000
--- a/src/rand.c
+++ b/src/rand.c
@@@ -14,16 -14,13 +14,26 @@@ int main(int argc, char *argv[]
    return EXIT_FAILURE;
}

++<<<<< HEAD
+ int max = atoi(argv[1]);
+ if (max > RAND_MAX) {
+   fprintf(stderr, "Cannot handle <number> larger than %d
(%d)\n",
+         RAND_MAX, max);
+   return EXIT_FAILURE;
+ } else if (max < 2) {
+   fprintf(stderr, "<number> cannot be smaller than %d
(%d)\n",
+         2, max);
+   return EXIT_FAILURE;
+
+=====
+ char *endptr = NULL;
+ long int val = strtol(argv[1], &endptr, 10);
+ if (*endptr) {
+   fprintf(stderr, "Invalid argument(s)\n");
+   return EXIT_FAILURE;
+
+ }
+ int max = (int) val;
++>>>>> 8c4ceca59d7402fb24a672c624b7ad816cf04e08

    srand(time(NULL));
    int result = random_int(max)
```

You can see a few differences from the ordinary unified `diff` format described in [Chapter 3, Developing with Git](#). First, it uses `diff --cc` in the header to denote that it uses the compact combined format (it uses `diff --combined` instead if you used the `git diff -c` command). The extended header lines, such as `index 293c8fc,4b87d29..0000000`, take into account that there is more than one source version. The chunk header, `@@@ -14,16 -14,13 +14,26 @@@`, is modified (different from the one for the ordinary patch) to prevent people from trying to apply a combined diff as unified diff, for example, with the `patch -p1` command.

Each line of the `diff` command is prefixed by two or more characters (two in the most common cases of merging two branches): the first character tells about the state of the line in the first preimage (*ours*) as compared to the result, the second character tells about the other preimage (*theirs*), and so on. For example, `++` means that the line was not present in either of versions being merged (here, in this example, you can find it on the line with the conflict marker).

Examining differences is even more useful for checking the resolution of a merge conflict.

To compare the result (current state of the working directory) with the version from the current branch (merged into), that is, *ours* version, you can use `git diff --ours`; similarly, for the version being merged (*theirs*), and the common ancestor version (*base*).

How do we get there: `git log --merge`

Sometimes, we need more context to decide which version to choose or to otherwise resolve a conflict. One such technique is reviewing a little bit of history, to remember why the two lines of development being merged were touching the same area of code.

To get the full list of divergent commits that were included in either branch, we can use the triple-dot syntax that you learned in [Chapter 2, Exploring Project History](#), adding the `--left-right` option to make Git

show which side the given commit belongs to:

```
$ git log --oneline --left-right HEAD...MERGE_HEAD
```

We can further simplify this and limit the output to only those commits that touched at least one of the conflicted files, with a `--merge` option to `git log`, for example:

```
$ git log --oneline --left-right --merge
```

This can be really helpful in quickly giving you the context you need to help understand why something conflicts and how to more intelligently resolve it.

Avoiding merge conflicts

While Git prefers to fail to automerge in a clear way, rather than to try elaborate merge algorithms, there are a few tools and options that one can use to help Git avoid merge conflicts.

Useful merge options

One of the problems while merging branches might be that they use different end of line normalization or clean/smudge filters (see [Chapter 4, Managing Your Worktree](#)). This might happen when one branch added such configuration (changing `gitattributes` file), while the other did not. In the case of end of line character configuration changes, you would get a lot of spurious changes, where lines differ only in the EOL characters. In both cases, while resolving a three-way merge, you can make Git run a virtual check out and check in of all the three stages of a file. This is done by passing the `renormalize` option to the recursive merge strategy (`git merge -xrenormalize`). This would, as the name suggests, normalize end of line characters, and make them the same for all stages.

Changing end of line can lead to what can be considered a part of *whitespace-related conflicts*. It's pretty easy to tell that it is the case while looking at the conflict, because every line is removed on one side and added again on the other, and `git diff --ignore-whitespace`

shows a more manageable conflict (or even a conflict that is resolved). If you see that you have a lot of whitespace issues in a merge, you can abort and redo it, but this time, with `-Xignore-all-space` or `-Xignore-space-change`. Note that whitespace changes mixed with other changes to a line are not ignored.

Sometimes, mismerges occur due to unimportant matching lines (for example, braces from distinct functions). You can make Git spend more time minimizing differences by selecting *patience diff algorithm* with `-Xpatience` or `-Xdiff-algorithm=patience`.

If the problem is misdetected renames, you can adjust the rename threshold with `-Xrename-threshold=<n>`.

Rerere – reuse recorded resolutions

The **rerere** (reuse recorded resolutions) functionality is a bit of a hidden feature. As the name of the feature implies, it makes Git remember how each conflict was resolved chunk by chunk, so that the next time Git sees the same conflict it would be able to resolve it automatically. Note, however, that Git will stop at resolving conflicts and that it does not autocommit the said rerere-based resolution, even if it resolves it cleanly (if it is superficially correct).

Such a functionality is useful in many scenarios. One example is the situation when you want a long-lived (long development) branch to merge cleanly at the end, but you do not want to create intermediate merge commits. In this situation, you can do trial merges (merge, then delete merge), saving information about how merge conflicts were resolved to the `rerere` cache. With this technique, the final merge should be easy, because most of it would be cleanly resolved from the resolutions recorded earlier.

Another situation you can make use of the `rerere` cache, is when you merge a bunch of topic branches into a testable permanent branch. If the integration test for a branch fails, you would want to be able to rewind the failed branch, but you would rather not lose the work spent on

resolving a merge.

Or perhaps, you have decided that you rather use rebase than merge. The rerere mechanism allows us to translate the merge resolution to the rebase resolution.

To enable this functionality, simply set `rerere.enabled` to true, or create the `.git/rr-cache` file.

Dealing with merge conflicts

Let's assume that Git was not able to autmerge cleanly, and that there are merge conflicts that you need to resolve to be able to create a new merge commit. What are your options?

Aborting a merge

First, let's cover how to get out of this situation. If you weren't perhaps prepared for conflicts or if you do not know enough about how to resolve them, you can simply back out from the merge you started with `git merge --abort`.

This command tries to reset to the state before you started a merge. It might be not able to do this if you have not started from a clean state. Therefore it is better to stash away changes, if there are any, before performing a merge operation.

Selecting ours or theirs version

Sometimes, it is enough to choose one version in the case of conflicts. If you want to have all the conflicts resolved this way, forcing all the chunks to resolve in favor of the *ours* or *theirs* version, you can use the `-Xours` (or `-Xtheirs`) option or the recursive merge strategy. Note that `-Xours` (merge option) is different from `-s ours` (merge strategy); the latter creates a fake merge, where the merge contents are the same as the *ours* version, instead of taking *ours* version only for conflicted files.

If you want to do this only for selected files, you can recheckout the file

with the ours or theirs version with `git checkout --ours / --theirs`.

You can examine the base, ours, or theirs version with `git show :1:file, :2:file, :3:file`, respectively.

Scriptable fixes – manual file remerging

There are types of changes that Git can't handle automatically, but they are scriptable fixes. The merge can be done automatically, or at least is much easier, if we could transform the "ours", "theirs" and "base" version first. Renormalization after changing how the file is checked out and stored in the repository (eol and clean/smudge filters) and handling the whitespace change are built-in options. Another non built-in example could be changing the encoding of a file, or other scriptable set of changes such as renaming variables.

To perform a scripted merge, first you need to extract a copy of each of these versions of the conflicted file, which can be done, with the `git show` command and a `:<stage>:<file>` syntax:

```
$ git show :1:src/rand.c >src/rand.common.c  
$ git show :2:src/rand.c >src/rand.ours.c  
$ git show :3:src/rand.c >src/rand.theirs.c
```

Now that you have in the working area the contents of all the three stages of the files, you can fix each version individually, for example with `dos2unix` or with `iconv`, and so on. You can then remerge the contents of the file with the following:

```
$ git merge-file -p \  
rand.ours.c rand.common.c rand.theirs.c >rand.c
```

Using graphical merge tools

If you want to use a graphical tool to help you resolve merge conflicts, you can run `git mergetool`, which fires up a visual merge tool and guides invoked tool through all the merge conflicts.

It has a wide set of preconfigured support for various graphical merge helpers. You can configure which tool you want to use with `merge.tool`.

If you don't do this, Git would try all the possible tools in the sequence which depends on the operating system and the desktop environment.

You can also configure a set up for your own tool.

Marking files as resolved and finalizing merges

As described earlier, if there is a merge conflict for a file, it will have three stages in the index. To mark a file as resolved, you need to put the contents of a file in stage 0. This can be done by simply running `git add <file>`.

When all the conflicts get resolved, you need to simply run `git commit` to finalize the merge commit (or you can skip marking each file individually as resolved and just run `git commit -a`). The default commit message for merge summarizes what we are merging, including a list of the conflicts if any, and adds a shortlog of the merged-in branches by default. The last is controlled by the `--log` option and the `merge.log` configuration variable.

Resolving rebase conflicts

When there is a problem with applying a patch or a patch series, cherry-picking or reverting a commit, or rebasing a branch, Git will fall back to using the three-way merge algorithm. How to resolve such conflicts is described in earlier sections.

However, for some of these methods, such as rebase, applying mailbox, or cherry-picking a series of commits, that are done stage by stage (a sequencer operation), there are other issues, namely, what to do if there is a conflict during such an individual stage.

You have three options. You can resolve the conflict, and continue the operation with the `--continue` parameter (or in case of `git am`, also `--resolved`). You can abort the operation and reset `HEAD` to the original branch with `--abort`. Finally, you can use `--skip` to drop a revision, perhaps because it is already present in the upstream and we can drop it during replaying.

git-imerge – incremental merge and rebase for git

Both rebase and merge have their disadvantages. With merge, you need to resolve one big conflict (though using test merges and rerere to keep up-to-date proposed resolutions could help with this) in an all-or-nothing fashion. There is almost no way to save partially a done merge or to test it; `git stash` can help, but it might be an inadequate solution.

Rebase, on the other hand, is done in step-by-step fashion. But it is unfriendly to collaboration; you should not rebase published parts of the history. You can interrupt a rebase, but it leaves you in a strange state (on an anonymous branch).

That's why the `git imerge` third-party tool was created. It presents conflicts pair wise in small steps. It records all the intermediate merges in such a way that they can be shared, so one person can start merging and the other can finish it. The final resolution can be stored as an ordinary merge, as an ordinary rebase, and as a rebase with history.

Summary

This chapter has shown us how to effectively join two lines of development together, combining commits they gathered since their divergence.

First, we got to know various methods of combining changes: merge, cherry-pick, and rebase. This part focused on explaining how these functionalities work at higher levels: at the level of the DAG of revisions. You learned how merge and rebase works, and what is the difference between them. Some of the more interesting uses of rebase, such as transplanting a topic branch from one long-lived branch to another, were also shown.

Then, you learned what to do in case Git is not able to automatically combine changes, that is, what can be done in the presence of a merge conflict. The important part of this process is to understand how the three-way merge algorithm works, and how the index and the working area are affected in case of conflicts. You now know how to examine failed merges and how to examine proposed resolutions, how to try avoiding conflicts, and finally how to resolve them and mark them as resolved.

The next chapter, *Keeping History Clean*, will explain why we might want to rewrite history to keep it clean (and what does it mean). One of the tools to rewrite history is an interactive rebase, a close cousin of an ordinary rebase operation described there. You will learn various methods of rewriting commits: how to reorder them, how to split them if they are too large, how to squash fix with the commit it is correcting, and how to remove a file from the history. You will find what you can do if you cannot rewrite history (understanding why rewriting published history is bad), but you still need to correct it: with the mechanisms of replacing and of notes. While at it, we will talk about other applications of these mechanisms.

Chapter 8. Keeping History Clean

The previous chapter, *Merging Changes Together*, described how to join changes developed by different people (as described in [Chapter 5, Collaborative Development with Git](#)), or just developed in a separate feature branch (as shown in [Chapter 6, Advanced Branching Techniques](#)). One of the techniques was rebase, which can help bring a branch to be merged to a better state. But if we are rewriting history, perhaps it would be possible to also modify the commits being rebased to be easier for review, making the development steps of a feature clearer? If rewriting is forbidden, can one make history cleaner without it? How do we fix mistakes if we cannot rewrite history?

This chapter will answer all those questions. It will explain why one might want to keep clean history, when it can and should be done, and how it can be done. Here you will find step-by-step instructions on how to reorder, squash, and split commits. This chapter will also describe how to do large-scale history rewriting (for example clean up after imports from other VCS) and what to do if one cannot rewrite history (how to use revert, replacements, and notes).

To really understand some of the topics presented here and to truly master their use, you need some basics of Git internals that are presented at beginning of this chapter.

In this chapter, we will cover the following topics:

- The basics of the object model of Git repositories
- Why you shouldn't rewrite published history and how to recover from it
- The interactive rebase: reordering, squashing, splitting, and testing commits
- Large-scale scripted history rewriting
- Reverting revision, reverting a merge, and remerging after reverted merge
- Amending history without rewriting with grafts and replacements

- Appending additional information to the objects with notes

An introduction to Git internals

To really understand and make good use of at least some of the methods described in this chapter, you would need to understand at least the very basics of Git internals. Among others, you would need to know how Git stores the information about revisions.

One would also require to know how to manipulate such data and how to do it from a script. Git provides a set of low-level commands to use in scripts, as a supplement to the user-facing high-level commands. These commands are very flexible and powerful, though perhaps not very user-friendly. Knowledge about this scripted interface will help us also administer the Git repositories via hooks in [Chapter 11, Git Administration](#).

Git objects

In [Chapter 2, Exploring Project History](#), you have learned that Git represents history as the Directed Acyclic Graph (DAG) of revisions, where each revision is a graph node represented as a **commit object**. Each commit is identified by a SHA-1 identifier. We can use this identifier (in full, or in an ambiguous shortened form) to refer to any given version.

The commit object consists of revision metadata, links to zero or more parent commits, and the snapshot of the project's files at the revision it represents. The revision metadata includes authorship (who and when made the changes), committership (who and when created the commit object), and of course the commit message.

It is interesting to see how Git represents the snapshot of project's files at the given revision. Git uses **tree objects** to represent directories, and **blob objects** (Binary Large OBject (**BLOB**)) to represent contents of a file. Besides the commit, tree, and blob objects, there might also be **tag objects** representing annotated and signed tags.

Each object is identified by the SHA-1 hash function over its contents or, to be more exact, over the type and the size of the object, plus its contents. Such a content-based identifier does not require a central naming service. Thanks to this fact, each and every distributed repository of the same project will use the same identifiers and we do not have to worry about name collisions:

```
# calculate SHA-1 identifier of blob object with Git
$ printf "foo" | git hash-object -t blob -stdin
# calculate SHA-1 identifier of blob object by hand
$ printf "blob 3\0foo" | shasum
```

We can say that the Git repository is the content-addressed object database. That's, of course, not all there is; there are also references (branches and tags) and various configurations, and other things.

Let's describe Git objects in more detail, starting bottom-up. We can examine objects with the low-level `git cat-file` command:

- **Blob:** These objects store the contents of the file at the given revision. Such an object can be created using the low-level `git hash-object -w` command. Note that, if different revisions have the same contents of a file, it is stored only once thanks to content-based addressing:

```
$ git cat-file blob HEAD:COPYRIGHT
Copyright (c) 2014 Company
All Rights Reserved
```

- **Tree:** These objects represent directories. Each tree object is a list of entries sorted by the filename. Each entry is composed of combined permissions and type, name of the file or directory, and a link (that is SHA-1 identifier) of an object connected with the given path, either the tree object (representing the subdirectory), the blob object (representing the file contents), or rarely the commit object (representing the submodule). Note that, if different revisions have the same contents of a subdirectory, it will be stored only once thanks to content-based addressing:

```
$ git cat-file -p HEAD^{tree}
100644 blob 862aaaf... COPYRIGHT
```

```
100644 blob 25c3d1b... Makefile
100644 blob bdf2c76... README
040000 tree 7e44d2e... src
```

Note that the real output includes full 40-character SHA-1 identifiers, not a shortened one, as shown in the preceding example. You can create tree objects out of the index (that you can create using the `git update-index` command) with `git write-tree`.

- **Commit:** These objects represent revisions. Each commit is composed of a set of headers (key-value data) that includes zero or more parent lines, and exactly one tree line with the link to the `tree` object representing a snapshot of the repository contents: the top directory of a project. You can create a commit with a given tree object as a revision snapshot by using the low-level `git commit-tree` command or by simply using `git commit`:

```
$ git cat-file -p HEAD
tree 752f12f08996b3c0352a189c5eed7cd7b32f42c7
parent cbb91914f7799cc8aed00baf2983449f2d806686
parent bb71a804f9686c4bada861b3fc3cfb5600d2a47
author Joe Hacker <joe@example.com> 1401584917 +0200
committer Bob Developer <bob@example.com> 1401584917 +0200
```

Merge remote branch 'origin/multiple'

- **Tag:** These objects represent annotated tags, of which signed tags are a special case. Tag objects also consist of a series of headers (among others link to the tagged object) and a tag message. You can create a tag object with a low-level `git mktag` command, or simply with `git tag`:

```
$ git cat-file tag v0.2
object 5d2584867fe4e94ab7d211a206bc0bc3804d37a9
type commit
tag v0.2
tagger John Tagger <john@example.com> 1401585007 +0200

random v0.2
```

Note

The Git internal format for the author, committer, and tagger dates is

`<unix timestamp> <timezone offset>`. The Unix timestamp (POSIX time) is the number of seconds since the Unix epoch, which is 00:00:00 Coordinated Universal Time (UTC), Thursday, 1 January 1970 (1970-01-01T00:00:00Z), not counting leap seconds. This denotes when the event took place. You can print the Unix timestamp with `date "%s"` and convert it into other formats with `date --date="@<timestamp>"`.

The timezone offset is a positive or negative offset from UTC in the HHMM (hours, minutes) format. For example, CET (that is 2 hours ahead UTC) is +0200. This can be used to find local time for an event.

Some Git commands work on any type of objects. For example, you can tag any type of objects, not only commits. You can, among others, tag a blob object to keep some unrelated piece of data in the repository and have it available in each clone. Public keys can be such data.

Notes and replacements, which will be described later in this chapter, also work on any type of objects.

The plumbing and porcelain Git commands

Git was developed in the bottom-up fashion. This means that its development started from basic blocks and built upward. Many user-facing commands were once built as shell scripts utilizing these basic low-level blocks to do their work. Because of this, we can distinguish between the two types of Git commands.

The better known types are the so-called **porcelain** commands, which are high-level, user-facing commands ("porcelain" is a play of words on calling *engine* level commands *plumbing*). The output of these commands is intended for the end user. This means that its output can be changed to be more user-friendly , and therefore its output can be different in different Git versions. with the Git version. The user (you) is smart enough to understand what happened, if presented, for example, with an additional information, or with a changed wording, or with a changed formatting.

This is not the case for the scripts that you may write (here, in this chapter, for example as a part of the scripted rewrite with `git filter-branch`). Here, you need unchanging output; well, at least, for the scripts that are used more than once (as hooks, as the `gitattribute` drivers, and as helpers). You can often find a switch, usually named `--porcelain`, that ensures the command output is immutable. For other commands, the solution is to specify the format fully. Alternatively, you can use low-level commands intended for scripting: the so-called **plumbing** commands. These commands usually do not have user-friendly defaults, not to mention do-what-I-mean-ness. Their output does not depend on the Git configuration; nor that many of them can be configured via Git configuration file.

The `git(1)` manpage includes a list of all the Git commands separated into porcelain and plumbing. The distinction between plumbing and porcelain commands was mentioned as a tip in [Chapter 4, Managing Your Worktree](#), when we encountered the first low-level plumbing command without a user-facing and user-friendly porcelain equivalent.

Environment variables used by Git

Git uses a number of shell environment variables to determine how it behaves. For user-facing porcelain commands that are shell scripts, they are used to pass data to the low-level plumbing commands doing the work, in addition to using standard input (pipelines) and command parameters.

Occasionally, it comes in handy to know what these environment variables are and how they can be used to make Git behave the way you want it to. This will be very visible in the section about scripted history rewriting with `git filter-branch` (especially, `--env-filter`) later in this chapter.

Environment variables fall between the Git configuration and command parameters in priority: environment variables overriding configuration and command parameters overriding environment variables. Well, except for fallback non-Git-specific environment variables, such as

`PAGER` and `EDITOR`, that take lowest precedence and can be overridden by configuration variables, such as `core.pager` and `core.editor`.

What follows is not meant to be an exhaustive list of all the environment variables that Git uses, but only a selected set of the ones especially useful and connected to the topic of this chapter.

Environment variables affecting global behavior

Some of Git's general behavior as a whole (the paths it searches and the external programs its uses) depends on environment variables.

`GIT_EXEC_PATH` determines where the core `git` programs are installed. You can check and set the current setting with `git --exec-path`. The default value is set at installation.

Where Git looks for configuration files is also affected by environment variables. The user-specific configuration file (also called the *global* configuration file) can be found either at `$XDG_CONFIG_HOME/git/config` (or `$HOME/.config/git/config` if `XDG_CONFIG_HOME` is not set or empty) or at `$HOME/.gitconfig`, values in the latter file taking precedence. You can override the values of either of these variables, namely `HOME` and `XDG_CONFIG_HOME` (which many other things depend on) in a shell profile for a truly portable Git installation.

The location of the system-wide configuration file is set at installation (usually, `/etc/gitconfig`), but you can skip reading from this file by setting the `GIT_CONFIG_NOSYSTEM` environment variable. This can be useful if your system configuration is interfering with your commands. Or, you can set the single configuration file to be used by Git with `GIT_CONFIG` environment variable (for `git config` this is equivalent to using the `--file` option).

The `GIT_PAGER` environment variable controls the program used to display a multipage output on the command line if the standard output is a terminal. If this is unset, then the `core.pager` configuration variable, the `PAGER` environment variable, and the built-in value—that is, the `less` program—will be used as a fallback, whichever is set, in this order.

A similar situation exists with `GIT_EDITOR` which is used to configure the editor to launch in the interactive mode when the user needs to edit some text (a commit message, for example). Note that the editor can be a script that is generating required output, instead of a real editor. The fallback environment variable is `EDITOR` or `VISUAL`, depending on the environment (after `core.editor`).

Environment variables affecting repository locations

Git uses several environment variables to determine how it interfaces with the current repository. These environment variables apply to all the core Git commands.

The `GIT_DIR` and `GIT_WORK_TREE` environment variables specify, if set, the location of the `.git` directory (the administrative area containing the repository) and its work tree (working area), respectively. The `--git-dir` command-line option can also be used to set the location of the repository. The location of the work tree can be controlled by the `--work-tree` option and the `core.worktree` configuration variable; well, if the repository is not bare and there exists a working area. By default, the repository that ends in `/ .git` is considered to be not bare; you can also set the `core.bare` configuration variable explicitly.

If the location of the repository is not set explicitly, Git will start from the current directory walk up the directory tree, looking for a `.git` directory at each step. If it finds it, the directory it was in when it found `.git` becomes the work tree (the top directory of the project), and the found `.git` directory is the location of the repository. You can specify a set of directories as a colon-separated list of absolute paths where Git should stop this walk with `GIT_CEILING_DIRECTORIES` (for example, to exclude slow-loading network directories); by default, Git would stop only at filesystem boundaries (unless `GIT_DISCOVERY_ACROSS_FILESYSTEM` is set to `true`).

You can use the `GIT_INDEX_FILE` environment variable to specify the location of an alternate index file. The default is to use `$GIT_DIR/index`.

Note that the index is not present in bare repositories. This variable can be used to create or modify a state of commit without touching the working area, that is, without touching the filesystem.

The `GIT_OBJECT_DIRECTORY` variable can be used to specify the location of the object storage directory; the default is to use `$GIT_DIR/objects`.

Also, due to the immutable nature of `git` objects and the fact that they are content-addressed, old objects can be archived into shared, read-only directories and, which can be outside `GIT_OBJECT_DIRECTORY`. Of course you need to tell Git where to find them. Or, in other words, Git repositories can share the object database (with some caveats). See `git clone --reference <repository> <URL>`, for example. This issue is covered in detail in [Chapter 9, Managing Subprojects - Building a Living Framework](#).

You can use the `GIT_ALTERNATE_OBJECT_DIRECTORIES` environment variable to specify additional directories that can be used to search for the `git` objects. This variable specifies a ":" separated list of paths the object would be read from, in addition to what is in the `$GIT_DIR/objects/info/alternates` file and the repository's own object database. New objects will not be written to these alternates.

Tip

How to compare two local repositories?

Let's assume that you want to compare two local repositories, but for some reason, you cannot just add one as a remote of the other, and then fetch from that remote. One example of such restriction would be using a read-only storage.

Being in one of repositories, you can do the following:

```
GIT_ALTERNATE_OBJECT_DIRECTORIES=../repo/.git/objects \
git diff \
$(GIT_DIR=../repo/.git git rev-parse --verify HEAD) \
HEAD
```

For the other repository, you need to get the universal identifier for an object, that is, its SHA-1. You can do so with the `git rev-parse` command. To turn a reference, such as `HEAD` or `HEAD:README`, it needs to be run in the other repository, which can be done with either the `GIT_DIR` environment variable (as in the example) or the `--git-dir` command-line option.

Environment variables affecting committing

The final creation of a Git commit object is usually done internally by the `git commit-tree` plumbing command. While parent information is provided as command parameters on the command line and `git commit-tree` gets a commit message on standard input, author and committer information is taken from the following environment variables.

The `GIT_AUTHOR_NAME` and `GIT_COMMITTER_NAME` commands are the human-readable name in the `author` and `committer` fields, respectively. The e-mail address can be set with `GIT_AUTHOR_EMAIL` for the `author` field and `GIT_COMMITTER_EMAIL` for the `committer` field, with generic `EMAIL` environment variable used as a fallback if the configuration variable `core.email` is not set. `GIT_AUTHOR_DATE` is the timestamp used for the `author` field with a date in the RFC 2822 e-mail format (Fri, 08 May 2015 01:35:42 +0200), the ISO 8601 standard date format (2015-05-08T01:36:48+0200), the Git internal format that is Unix time plus +hhmm numeric time zone (1431041884 +0200), or the any other datetime format supported by Git; similar for `GIT_COMMITTER_DATE`.

In case, (some of) these environment variables are not set, the information is taken from the configuration or Git tries to guess it. If the required information is not provided, and Git cannot guess it, then commit will fail.

Rewriting history

Many times, while working on a project, you may want to revise your commit history. One reason for this could be to make it easier to review before submitting changes upstream. Another reason would be to take reviewer comments into account in the next improved version of changes. Or perhaps you'd like to have a clear history while finding regressions using bisection, as described in [Chapter 2, Exploring Project History](#).

One of the great things about Git is that it makes revising and rewriting history possible, while providing a wide set of tools to revise history and make it clean.

Note

There are two conflicting views among users of the version control system: one states that history is sacred and you should better show the true history of the development, warts and all, and another that states that you should clean up the new history for better readability before publishing it.

An important issue to note is that, even though we talk about rewriting history, objects in Git (including commits) are *immutable*. This means that rewriting is really creating a modified copy of commits, a new path in the DAG of revisions. Then appropriate branch reference is switched to point to the just created new path. The original, pre-rewrite commits are there in the repository, referenced and available from the reflog (and also, `ORIG_HEAD`). Well, at least, until they get pruned (that is, deleted) as unreferenced and unreachable objects during garbage-collecting. Though, this happens only after the reflog expires.

Amending the last commit

The simplest case of history rewriting is correcting the latest commit on a branch (the current commit).

Sometimes, you notice a typo (an error) in a commit message, or that you have committed incomplete change in the last revision. If you have not pushed (published) your changes, you can amend the last commit. This is done with the `--amend` option to `git commit`.

The result of amending a commit is shown in *Fig 6* in [Chapter 3, Developing with Git](#). Note that there is no functional difference between amending the last commit and changing the commits deeper in history. In both the cases you are creating a new commit, leaving the old version referenced by the reflog.

Here, the index (that is, the explicit staging area for commits) shows its usefulness again. For example, if you want to simply fix only the commit message, and you do not want to make any changes, you can use `git commit --amend` (note the lack of `-a` / `--all` option). This works even if you started work on a new commit; at least, assuming that you didn't add any changes to the index. If you did, you can put them away temporarily with `git stash`, fix the commit message of the last commit, and then pop stashed changes and restore the index with `git stash pop --index`.

If, on the other hand, you realize that you have forgotten some changes, you can just edit the files and use `git commit --amend --all`. And if the changes are interleaved, you can use `git add`, or its interactive version (utilizing knowledge from [Chapter 5, Managing Your Worktree](#)), to create the contents you want to have, finalizing it with `git commit --amend`.

An interactive rebase

Sometimes, you might want to edit commit deeper in history, or reorganize commits into a logical sequence of steps. One of the built-in tools that you can use in Git for this purpose is `git rebase --interactive`.

Here, we will assume that you are working on a feature using a separate topic branch and a topic branch workflow described and recommended

in [Chapter 6, Advanced Branching Techniques](#). We will also assume that you are doing the work in the series of logical steps, rather than in one large commit.

While implementing a new feature, you usually don't do it perfectly from the very beginning. You would want to introduce it in a series of self-contained small steps (see [Chapter 12, Git Best Practices](#)) to make code review (or code audit) and bisection (finding the cause of regressions) easier. Often only after finishing work you see how to split it better. It is also unreasonable to expect that you would not make mistakes while implementing a new feature.

Before submitting the changes (either pushing to a central repository, pushing to your own public repository and sending pull requests, or using some other workflows described in [Chapter 5, Collaborative Development with Git](#)), you would often want to update your branch to the current up-to-date state of a project . By rebasing your changes on top of current state, and having them up to date, you would make it easier for the maintainer (the integration manager) to ultimately merge your changes in, when they are accepted for the inclusion in the mainline. Interactive rebase allows you to clean up history, as described earlier, while doing it.

Besides tidying up before publishing changes, there is also additional use for tools such as an interactive rebase. While working on a more involved feature, the very first submission is not always accepted into an upstream and added to the project. Often, patch review finds problem with the code or with the explanation of the changes. Perhaps, something is missing (for example, the feature lacks documentations or tests), some commit needs to be fixed, or the submitted series of patches (or a branch submitted in a pull request) should be split into smaller commits for easy review. In this case, you would also use an interactive rebase (or an equivalent tool) to prepare a new version to submit, taking into account the results of code inspection.

Reordering, removing, and fixing commits

Rebase, as described in [Chapter 7, Merging Changes Together](#), consists of taking a series of changes of the commits being rebased and reapplying them on top of a new base (a new commit). In other words, rebase moves changesets, not snapshots. Git starts the interactive rebase by opening the instructions sheet corresponding to those series operations of reapplying changes in an editor.

Note

You can configure the text editor used for editing the rebase instruction file separately from the default editor (used, for example, to edit commit message) with the `sequence.editor` configuration variable, which in turn can be overridden by the `GIT_SEQUENCE_EDITOR` environment variable.

Like in the case of the template for editing commits, the instruction sheet is accompanied by the comments explaining what you can do with it (note that if you use older Git version, some interactive rebase commands might be missing):

```
pick 89579c9 first commit in a branch
pick d996b71 second commit in a branch
pick 6c89dee third commit in a branch

# Rebase 89579c9..6c89dee onto b8ffffe1
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log
message
# x, exec = run command (the rest of the line) using shell
# d, drop = remove commit
#
# These lines can be re-ordered; they are executed from top to
bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be
```

```
aborted.  
#  
# Note that empty commits are commented out
```

As explained in the comments, the instructions are in the order of execution, starting from the instruction on the top to create the first commit with the new base as its parent, and ending with the instruction copying commit at the tip of the branch being rebased at the bottom. This means that revisions are listed in an increasing chronological order, older commits first. This is the reverse order as compared to the `git log` output with the most recent commit first (unless using `git log --reverse`). This is quite understandable; the rebase reapplies changesets in the order they were added to the branch, while the log operation shows commits in the order of reachability from the tips.

Each line of the instruction sheet consists of three elements separated by spaces. First, there is a one-word command; by default, the interactive rebase starts with `pick`. Each command has a one-letter shortcut that you can use instead of the long form, as shown in the comments (for example you can use "`'p'`" in place of "`'pick'`").

Next, there is a uniquely shortened SHA-1 identifier of a commit to be used with the command. Strictly speaking, it is the identifier of a commit being rebased, which it had before the rebase started. This shortened SHA-1 identifier is used to pick the appropriate commit (for example while reordering lines, which means reordering commits).

Last, there is the description (the subject) of a commit. It is taken from the first line of the commit message (specifically, it is the first paragraph of the commit message with the line breaks removed, where a paragraph is defined as the set of subsequent lines of text separated from other paragraphs by at least one empty line—that is, two or more end-of-line characters). This is one of the reasons why the first line of the commit message should be a short description of changes (see [Chapter 12, Git Best Practices](#)). This description is for you to help decide what to do with the commit; Git uses its SHA-1 identifier and ignores the rest of the line.

Reordering commits with the interactive rebase is as simple as reordering lines in the instruction sheet. Note, however, that if the changes were not independent, you might need to resolve conflicts, even if they would be no merge conflicts without doing reordering. In such cases, as instructed by Git, you would need to fix conflicts, mark conflicts as resolved, (for example, with `git add`), and then run `git rebase --continue`. Git will remember that you are in the middle of an *interactive* rebase, so you don't need to repeat the `--interactive` option.

The other possibility of dealing with a conflict, namely, skipping a commit, rather than resolving a conflict, by running `git rebase --skip`, is here as well. By default, rebase removes changes that are already present in upstream; you might want to use this command in case the rebase doesn't detect correctly that the commit in question is already there in the branch we are transplanting revisions onto. In other words, do skip a commit if you know that the correct resolution of a conflict is an empty changeset.

Note

You can also make Git present you again with the instruction sheet at any time when rebase stops for some reason (including an error in the instruction sheet, like using the `squash` command with the first commit) with `git rebase --edit-todo`. After editing it, you can continue the rebase.

To **remove changes**, you simply need to remove the relevant line from the instruction sheet, or to comment it out, or -- with the newest Git -- use the `drop` command. You can use it to drop failed experiments, or to make it easier on the rebase by deleting changesets that you know are already present in the rebase onto the upstream, though perhaps in a different form. Note, though, that removing the instruction sheet altogether aborts the rebase.

To **fix a commit**, change the `pick` command preceding the relevant commit in the instruction sheet to `edit` (or just `e`). This would make

rebase stop at this commit, that is, at the step of reapplying changes, similar to the case with a conflict. To be precise, the interactive rebase applies the commit in question, so it is the `HEAD` commit and stops the process giving control to the user. You can then fix this commit, as if it were a current one with `git commit --amend`, as described in *Amending the last commit*. After changing it to your liking, run `git rebase --continue`, as explained in the instruction that Git prints.

Note

A proper `git-aware` command-line prompt, such as the one from the `Git contrib` command, would tell you if you are in the middle of the rebase (see [Chapter 10, Customizing and Extending Git](#)). If you are not using such a prompt, you can always check what's happening with `git status`:

```
$ git status
rebase in progress; onto b3cebef
You are currently rebasing branch 'subsys' on 'b3cebef'.
  (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)
```

As you can see, you can always go to the state before starting the rebase with `git rebase --abort`.

If you only want to change the commit message (for example, to fix spellings or include additional information), you can skip the need to run `git commit --amend` and then `git rebase --continue` by using `reword` (or `r`) instead of `edit`. Git would then automatically open the editor with the commit message. Saving changes and exiting the editor will commit the changes, amend the commit, and continue the rebase.

Squashing commits

Sometimes, you might need to make one commit out of two or more, squashing them together. Maybe, you decided that it didn't make sense to split the changes and they are better together.

With the interactive rebase, you can reorder these commits, as needed, so they are next to each other. Then, leave the `pick` command for the first of the commits to be concatenated together (or change it to `edit`). For the rest of the commits, use the `squash` or `fixup` command. Git will then accumulate the changes and create the commit with all of them together. The suggested commit message for the folded commit is the commit message of the first commit with the messages of the commits with the `squash` command appended; commit messages with the `fixup` command are omitted. This means that the `squash` command is useful while squashing changes, while `fixup` is useful for adding fixes. If the commits had different authors, the folded commit will be attributed to the author of the first commit. The committer will be you, the person performing the rebase.

Let's assume that you noticed that you forgot to add some part of the changes to the commit. Perhaps, it is missing tests (or just negative tests) or the documentation. The commit is in the past, so you cannot just add to it by amending. You could use an interactive rebase or the patch management interface to fix it, but often it is more effective to create the commit with forgotten changes and squash it later.

Similarly, when you notice that the commit you created a while ago has a bug, instead of trying to edit it immediately, you can create a `fixup` commit with a bugfix to be squashed later.

If you are using this technique, some time might pass between noticing the need to append new changes or fix a bug and creating an appropriate commit, and the time taken to rebase. How to mark the said commit to squash or fixup? If you use the commit message beginning with the magic string `squash! ...` or `fixup! ...`, respectively, preceding the description (the first line of the commit message that is sometimes called **subject**) of a commit to be squashed into, you can ask Git to autosquash them, thus automatically modifying the to-do list of `rebase -i`. You can request this on an individual basis with the `--autosquash` option or you can enable this behavior by default with the `rebase.autoSquash` configuration variable. To create the appropriate magic commit message, you can use `git commit --squash/--fixup`. (with commit to be

squashed into / commit to be fixes as a parameter to this option)

Splitting commits

Sometimes, you might want to make two commits or more out of one commit, splitting it in two or more parts. You may have noticed that the commit is too large, perhaps it tries to do too much, and should be split in two. Or perhaps, you have decided that some part of a changeset should be moved from one commit to another, and extracting it into a separate commit is a first step towards that.

Git does not provide a one-step built-in command for this operation. Nevertheless, splitting commits is possible with the clever use of the interactive rebase.

To split a given commit, first mark it with the `edit` action. As described earlier, Git will stop at the specified commit and give the control back to the user. In the case of splitting a commit, when you return the control to Git with `git rebase --continue`, you would want to have two commits in place of one.

The problem of splitting a commit is comparable to the problem of having different changes tangled together from [Chapter 3, Developing with Git](#) (the section about interactive commit). The difference is that the commit is already created and copied from the branch being rebased. It is simple to fix it with `git reset HEAD^`; as described in [Chapter 4, Managing Your Worktree](#), this command will keep the working area at the (entangled) state of the commit to be split while moving the `HEAD` pointer and the staging area for the commit to the state before this revision. Then you can interactively add to the index the changes that you want to have in the first commit, composing the intermediate step in the staging area. Next, check whether you have what you want in the index, then create a commit from it using `git commit` without the `-a` / `--all` option. Repeat these last two steps as often as necessary.

Alternatively, instead of adding changes interactively, you can interactively remove changes to create the intermediate state for split

commit. This can be done with interactive reset, mentioned in [Chapter 4, Managing Your Worktree](#).

For the last commit in the series (the second one if you are splitting the commit in two), you can either add everything to the index making a working copy clean and create a commit from the index, or you can create a commit from the state of the working area (`git commit --all`). If you want to keep, or start from, the commit message of the original commit to be split, you can provide it with the `--reuse-message=<commit>` or `--reedit-message=<commit>` option while creating a commit. I think, the simplest way of naming a commit that was split (or that is being split) is to use reflog—it will be the `HEAD@{n}` entry just before reset: moving to `HEAD^` in the `git reflog` output.

Instead of crafting the commit in the staging area (in the index) starting from the parent of the commit to be split, and adding changes, perhaps interactively, you could start from the final state (that is, the commit to be split) and remove the changes that are to be in second step, for example, with `git reset --patch HEAD^` (interactive removal). Frankly, you can use any combination of techniques from [Chapter 4, Managing Your Worktree](#). I find, for example, graphical commit tools such as `git gui` quite useful (you can find what are the graphical commit tools, and their examples in [Chapter 11, Customizing and Extending Git](#)).

If you are not absolutely sure that the intermediate revisions you are creating in the index are consistent (they compile, pass the test suite, and so on), you should use `git stash save --keep-index` to stash away the not-yet-committed changes, bringing the working area to the state composed in the index. You can then test the changes (for example by using the testsuite), and if fixes are necessary amend the staging area. Alternatively, you can create the commit from the index and use plain `git stash` to save the state of the working area after each commit. You can then test and amend the created intermediate commit if the fixes are necessary. In both the cases, you need to restore changes with `git stash pop` before you work on a new commit in the split.

Testing each rebased commit

A good software development practice is to test each change before committing it. But it does not always happen. Let's assume that you forgot to test some commits or skipped it because the change seemed trivial and you were pressed for time. The interactive rebase allows you to execute tests (to be precise, any command) during the rebase process by adding the `exec (x)` action with an appropriate command after the commit you want to test.

The `exec` command launches the command (the rest of the line) in a shell: the one specified in `SHELL` environment variable, or the default shell if `SHELL` is not set. This means that you can use shell features (for POSIX shell, it would be using `cd` to change directories, "`>`" to command output redirection, and "`;`" and "`&&`" to sequence multiple commands, and so on). It's important to remember that the command to be executed is run from the root of the working tree, not from the current directory (the subdirectory you were in while starting the interactive rebase).

If you are strict about not publishing untested changes, you might have worried about the fact that rewritten commits, rebased on the top of the new changes, might not pass the tests, even if the originals have. You can, however, make the interactive rebase test each commit with the `--exec` option, for example:

```
$ git rebase --interactive --exec "make test"
```

This would modify the staring instruction sheet, inserting `exec make test` after each entry:

```
pick 89579c9 first commit in a branch
exec make test
pick d996b71 second commit in a branch
exec make test
pick 6c89dee third commit in a branch
exec make test
```

External tools – patch management interfaces

You might prefer fixing the old commit immediately at the time you have noticed the bug, and not postponing it till the branch is rebased.

The latter is usually done just before the branch is sent for review (to publish it). This might be quite some time after realizing the need to edit the past commit.

Git itself doesn't make it easy to fix the found bug straight away, not with built-in tools. You can, however, find third-party external tools that implement the patch management interface on the top of Git. Examples of such tools include **Stacked Git (StGit)** and **Git Quilt (Guilt)**.

These tools provide similar functionality to Quilt (that is, pushing/popping patches to/from a stack). With them, you have a set of work-in-progress *floating* patches in the Quilt-like stack. You have also accepted changes in the form of proper Git commits. You can convert between patch and commit and vice versa, move and edit patches around, move and edit commits (that is done by turning the commit and its children into patches and back again), squash patches, and so on.

This is, however, an additional tool to install, additional set of operations to learn (even if they make your work easier), and additional set of complications coming from the boundary between the Git and the tool in question. An interactive rebase is powerful enough nowadays and, with autosquash, the need for another layer on top of Git is lessened.

Scripted rewrite with the `git filter-branch`

In some cases, you might need to use more powerful tools than the interactive rebase to rewrite and clean up the history. You might want something that would rewrite the full history, and would do the rewrite noninteractively, given some specified algorithm to do the rewrite. Such situations are the task for `git filter-branch`.

The calling convention of this command is rather different than for the interactive rebase. First, you need to give it a branch or a set of branches to rewrite, for example, `--all` to rewrite all the branches. Strictly speaking, you give it the `rev-list` options as arguments, that is, a series of positive and negative references (see [Chapter 2, Exploring Project History](#) for definition). The command will only rewrite the positive refs

mentioned in the command line. This means that positive references, which are the upper limits of revision ranges, need to be able to be rewritten—to be branch names. Negative revisions are used to limit what is ran through the rewriting process; you can, of course, also specify a pathspec on a command line to limit the changes.

This command rewrites the Git revision history by applying custom filters (scripts) on each revision to be rewritten. That's another difference: rebase works by reapplying changesets, while filter-branch works with snapshots. One of the consequences of this is that, for the filter-branch, a merge is just a kind of a commit object, while the rebase skips merges, unless you use the `--preserve-merges` option that does not work well combined with the interactive mode.

And, of course, with the filter-branch, you use scripts for rewrite (that are called **filters**), instead of rewriting interactively: editing instruction sheets and running commands by hand to edit, reword, squash, split, or test commits during the rebase process. This means that the speed of the filter-branch operation is not limited by the speed of the user interaction, but by I/O. It is recommended to use an off-disk temporary directory for rewriting (if the filter requires it) with the `-d <directory>` option.

Note

Because `git filter-branch` is usually used for massive rewrites, it saves the original refs, pointing to the pre-rewritten history in the `refs/original/` namespace (you can override it with the `--original <namespace>` option).

The command would also refuse to start, unless forced, if there are already existing refs starting with `refs/original/`, or if there is anything in a temporary directory.

Running the filter-branch without filters

If you specify no filters, the commits will be recommitted without any changes. Such usage would normally have no effect, but it is permitted to allow in the future to compensate for (to fix) some Git bugs.

It is important to note that this command respects both *grafts* (it honors `.git/info/grafts` file) and *replacements* (refs in the `refs/replace/` namespace), though you can ask Git with a command line option to not follow the latter. Grafts and replacements are techniques to affect the history (or a rather a view of it) without rewriting any revisions. Both will be explained later in the *Replacements mechanism* section.

This means that running `git filter-branch` without any filter can be used to make permanent the effects of grafts or replacements by rewriting the selected commits. This way, you can use the following technique: use `git replace` on the specified commits to alter the view of a history, ensure that it looks correct (like you wanted it to look like), and then make the modification permanent with a filter-branch.

Additionally, while rewriting commits, `git filter-branch` respects the current value of a few relevant configuration variables. The values of those variables might have changed since the original creation of the commits being rewritten. This feature might be used, for example, to fix history if you have used nonstandard encoding for the commit messages (not UTF-8), but forgot to set `i18n.commitEncoding`. Rewriting history with no filters, with '`i18n.commitEncoding`' set correctly at that time, will nevertheless add the `encoding` header to the commit objects.

Available filter types for filter-branch and their use

There is a large set of different types of possible filters to specify how to rewrite history. You can specify more than one type of filter; they are applied in the listed order. Note that different filters have different performance considerations.

The command argument is always evaluated in the shell context and is called once per commit undergoing the rewrite. Information about a pre-rewrite SHA-1 identifier of a current commit (that is, the commit being rewritten) is passed to the filter using the `GIT_COMMIT` environment variable. In addition, there is a `map` shell function available that takes the original SHA-1 of a commit as an argument, and outputs either the rewritten or original SHA-1 depending on whether the commit was

rewritten or not at the time this shell function was invoked.

Also, `GIT_AUTHOR_NAME`, `GIT_AUTHOR_EMAIL`, `GIT_AUTHOR_DATE`, `GIT_COMMITTER_NAME`, `GIT_COMMITTER_EMAIL`, and `GIT_COMMITTER_DATE` are taken from the current commit and exported to the environment to make it easier to write the contents of the filter, and to affect the author and committer identities of the replacement commit. The filter-branch command uses `git commit-tree` to create a replacement commit if the filter function succeeds; if the command returns a nonzero exit status, then the whole rewrite will get aborted.

When writing filter scripts, just like for normal scripts, it is usually better to use low-level plumbing commands, rather than high-level porcelain commands designed for interactive use. In particular, the filter-branch command uses plumbing itself... without all the do what I mean (DWIM) niceties (like following `gitignore` files). If you prefer, though, you can use programs for filters, instead of shell scripts.

The `git filter-branch` command supports the following types of filters:

- `--env-filter`: This may be used to modify environments in which a commit is performed. You might use it to change author or committer information, namely their name, e-mail, or time of operation. Note that variables need to be re-exported.
- `--tree-filter`: This may be used to rewrite the contents of the commit, that is, the tree object the commit refers to. The command is evaluated in the shell, with the working directory set to the root of the project and current commit checked out. After the command finishes, the contents of the working area are used *as-is*, new files are auto-added, and disappeared files are auto-removed without considering any ignore rules (for example, from `.gitignore`).
- `--index-filter`: This may be used to rewrite the index and the staging area from which the rewritten commit will be created. It is similar to the tree filter, but is much faster, because it doesn't need to check out files into the working area (into the filesystem).
- `--parent-filter`: This may be used to rewrite the commit's parent

list. It receives a parent string in the form of the parent's command-line parameters to the `git commit-tree` command (`-p <parent full SHA-1>`) on a standard input, and shall output a new parent string on a standard output.

- `--msg-filter`: This may be used to rewrite the commit messages. It receives the original commit message on a standard input, and shall output a new commit message on a standard output.
- `--commit-filter`: This may be used to specify the command to be called instead of `git commit-tree`. This means getting `<tree> [(-p <parent>) ...]` as arguments to the filter command, and getting the log message on the standard input.

You can use in this filter a few convenience functions: `skip_commit "$@"` to leave out the current commit (but not its changes!), and `git_commit_non_empty_tree "$@"` to automatically skip no-change commits.

Note

`"$@"` expands to the positional parameters of the command starting from one. When the expansion occurs within double quotes, each parameter expands to a separate word. This is a standard POSIX shell feature, and can be used to pass all the parameters down unchanged.

- `--tag-name-filter`: This may be used to rewrite tag names. The original tag name is passed on a standard input, and the command shall write a new name to a standard output. The original tags are not deleted, but can be overwritten; use `--tag-name-filter cat` to simply update tags (stripping signatures).

Note that the signature gets stripped, because by definition, it is impossible to preserve them. Tags with rewritten names are properly rewritten to point to the changed object. Currently, there is no support to change the tagger, timestamp, tag message, or re-signing tags.

- `--subdirectory-filter <directory>`: This may be used to leave only the history of the given directory, and make this directory a

project root. Can be used to change a subdirectory of a project into a subproject; see also [Chapter 9: Managing Subprojects - Building a Living Framework](#).

Note that if you use the `git log` / `git rev-list` options to limit the set of revisions to rewrite (for example, `--all` to rewrite all the branches), you must separate them with "—" from the specification of filters and other `git filter-branch` options.

Examples of using the `git filter-branch`

Let's assume that you committed a wrong file to a repository by mistake and you want to **remove the file from the history**. Perhaps this was a site-specific configuration file with passwords or their equivalent.

Perchance, during "`git add .`", you have included a generated file that was not properly ignored (maybe it was a large binary file). Or mayhap, it turned out that you don't have the distribution rights to a file and you need to have it removed to avoid copyright violation.

Now you need to remove it from the project. Using `git rm --cached` would remove it only from future commits. You can also quite easily remove the file from the latest version by amending the commit (as described earlier in this chapter).

To excise the file from the entire history, (let's assume it is called `passwords.txt`), you can use `git filter-branch` with the `--tree-filter` option:

```
$ git filter-branch --tree-filter 'rm -f passwords.txt' HEAD
Rewrite fdfb73095fc0d594ff8d7f507f5fc3ab36859e3d (32/32)
Ref 'refs/heads/master' was rewritten
```

There is, however, a faster alternative—instead of using a tree filter, which involves writing out files, you can use delete files from the index using `git rm --cached` with the index filter. You need to ensure that the filter runs successfully and does not exit even if there are no files to delete; there is also no need for output:

```
$ git filter-branch --index-filter \
```

```
'git rm -f --cached -q --ignore-unmatch passwords.txt'  
HEAD
```

Or, you can use the BFG Repo-Cleaner third-party tool described in a later section.

You can use a filter branch to **remove all the specific types of commits** from the history, for example, commits by a specific author (one that, for example, didn't fulfill the copyright obligations, such as the contributor agreement). Note, however, that there is a very important difference between removing commits with filter-branch and removing them using a interactive rebase. A filter-branch removes nodes in the DAG of revisions, but does not remove the changes—there is simply no longer an intermediate step between two snapshots, and changes move to the child commit. On the other hand, an interactive rebase removes both commit and changes. This means that all the child commits are modified so that their snapshot does not include removed changes.

To remove a commit, you can use the `skip_commit` shell function in a commit filter:

```
$ git filter-branch --commit-filter '  
if [ "$GIT_AUTHOR_NAME" = "Bad Contributor" ];  
then  
    skip_commit "$@"  
else  
    git commit-tree "$@"  
fi' HEAD
```

You can use a filter-branch to permanently **join two repositories**, connect histories, and split the history in two. You can do this directly with a parent filter. For example to join repositories, making the commit `<root-id>` from the history of one of repositories being joined have `<graft-id>` commit (from the other repository) as a parent, you can use:

```
$ git filter-branch --parent-filter \  
'test "$GIT_COMMIT" = <root-id> && echo "-p <graft-id>" || cat'  
HEAD
```

You can **split history** at a given commit in two in a similar way, by

setting parents to an empty set with `echo ""`.

If you know that you have only one root commit (only one commit with no parents), you can simplify the method to join the history to the following command:

```
$ git filter-branch --parent-filter 'sed "s/^$/-p <graft  
id>/"' HEAD
```

In my opinion, however, it is simpler to use grafts or replacements, check whether the joined/split history renders correctly, and then make replacements permanent by running `filter-branch` without filters with the revision range starting, at least, from the rewritten joint/root commit. Still, the `--parent-filter` approach has an advantage if you can tell programmatically which revision or revisions to split (or join); a simple version of this technique is presented in the single-root join as shown in the preceding example.

Another common case is to **fix erroneous names or e-mail addresses in commits**. Perhaps, you forgot to run `git config` to set your name and e-mail address before you started working and Git guessed it incorrectly (if it couldn't guess it, it would ask for it before allowing the commit), and `.mailmap` is not enough. Maybe, you want to open the sources of the formerly proprietary closed-source program and need to change the internal corporate e-mail to a personal address.

In any case, you can change the e-mail addresses in the whole history with a `filter-branch`. You need to ensure that you are changing your commits. You can use `--env-filter` for this (though `--commit-filter` would work too, with just `git commit-tree "$@"` and no export lines):

```
$ git filter-branch --env-filter '  
if test "$GIT_AUTHOR_EMAIL" = "joe@localhost"; then  
    GIT_AUTHOR_NAME="Joe Hacker"  
    GIT_AUTHOR_EMAIL=joe@company.com  
    export GIT_AUTHOR_NAME GIT_AUTHOR_EMAIL  
fi' HEAD
```

This example presents a simplified solution, you would want to change

the committer data too, and the code is nearly identical.

If you are open-sourcing a project, you could also want to add the `Signed-off-by:` lines for the Digital Certificate of Origin (see [Chapter 12, Git Best Practices](#)):

```
$ git filter-branch --msg-filter \
'cat && echo && echo "Signed-off-by: Joe Hacker
<joe@company.com>"' \
HEAD
```

Suppose that you have noticed **a typo in the name of a subdirectory**, for example, `inlude/` instead of `include/`. If there is no problem rewriting it, you could fix it by using `--tree-filter` with `mv -f inlude include;` but with some ingenuity, we can use `--index-filter` faster:

```
$ git filter-branch --index-filter '
git ls-files --stage |
sed -e "s!\\(\t\"*\")inlude/!\\1include/!" |
GIT_INDEX_FILE=$GIT_INDEX_FILE.new \
    git update-index --index-info &&
mv "$GIT_INDEX_FILE.new" "$GIT_INDEX_FILE"
' HEAD
```

The explanation is as follows: we use the fact that the output of `git ls-files --stage` matches the format of input for `git update-index --index-info` (the latter command is a plumbing command underlying the `git add` porcelain). To replace text and fix a typo in a path name, the `sed` (streaming editor) utility is used. Here, we needed to write the regular expression to take care that some file names may require quoting. A temporary index file is used to make an atomic operation.

Often, some part of a larger project takes life on its own and it begins to make sense to use it separate from the project it started in. We would want to extract the history of this part to make its **subdirectory the new root**. To rewrite history in this way and discard all the other history, you can run:

```
$ git filter-branch --subdirectory-filter lib/foo -- --all
```

Though, perhaps, a better solution would be to use a specialized third-party tool, namely, `git subtree`. This tool (and its alternatives) will be discussed in [Chapter 9, Managing Subprojects – Building a Living Framework](#).

External tools for large-scale history rewriting

The `git filter-branch` command is not the only solution for the large-scale rewriting of the project's history. There are other tools that might be easier to use, either because they include lots of predefined clean-up operations, or because they provide some level of interactivity with the ability for scripted rewrite (with read-evaluate-print loop (REPL), similar to interactive shells in some interpreted programming languages).

Removing files from the history with BFG Repo Cleaner

The BFG Repo Cleaner is a simpler, faster, and specialized alternative to using the `git filter-branch` command for cleansing bad data out of your Git repository history: removing files and directories and replacing text in files (for example, passwords with placeholders). It is faster than `filter-branch` for its area of application, because it can assume that we don't care where in the directory hierarchy the bad file is, only that we want it to be gone. Also, it can use multiple cores with parallel processing, and it doesn't need to fork and the `exec` shell to run filter script for each commit—BFG is written in Scala and uses JGit as a Git implementation.

BFG is simpler to use in typical use cases, because it provides a set of command-line parameters specialized for removing files and fixing them, such as `--delete-files` or `--replace-text`, a query language of sorts. It lacks the flexibility (often unnecessary one) of `filter-branch`, though.

One issue you need to remember is that BFG assumes that you have fixed the contents of your current commit.

Editing the repository history with reposurgeon

The reposurgeon was originally created to help clean up artefacts created by the repository conversion (migrating from one version control system to another). It relies on being able to parse, modify, and emit the command stream in the `git fast-import` format, which is nowadays a common export and import format among source control systems thanks to it being version control agnostic.

It can be used for history rewriting, including editing past commits and metadata, excising commits, squashing (coalescing) and splitting commits, removing files and directories from history, and splitting and joining history.

The major advantage `reposurgeon` has over using `git filter-branch` is that it can be run in two modes: either as an interactive interpreter, a kind of debugger/editor for history with command history and tab completion, and a batch mode to execute commands given as arguments. This allows to interactively inspect history and test changes, and then batch run them for all the revisions.

The disadvantage is having to install and then learn to use a separate tool.

The perils of rewriting published history

There is, however, a very important principle. Namely, that you should never (or, at least, not without a very, very good reason) rewrite published history, especially when it comes to commits that got pushed to the public repository, or were otherwise made public. What you can do is to change those parts of the graph of the revisions that are private.

The reason behind this rule is that rewriting published history could cause trouble for downstream developers, if they based their changes on revisions that got rewritten.

This means that it is safe to rewrite and rebuild those public branches that are explicitly stated and documented to be in flux, for example, as a way of showing work in progress (such as `pu`: proposed updates type of

branch). Another possibility for the safe rewriting of a public branch is to do it at specific stages of the project's life, namely, after creating a new release; again, this needs to be documented.

The consequences of upstream rewrite

Now, you will see on a simple example the perils of rewriting published history (for example, rebasing) and how it causes trouble. Let's assume that there are two public branches that are of interest: `master` and `subsys`. The latter branch is based on (forked from) the former. Let's also assume that a downstream developer (who might be you) created a new `topic` branch based on `subsys` for his/her own work, but did not published it yet; it is present only in his/her local repository. This situation is shown in *Fig 1* (the darker blue color denotes the revisions present only in the local repository of the downstream developer):

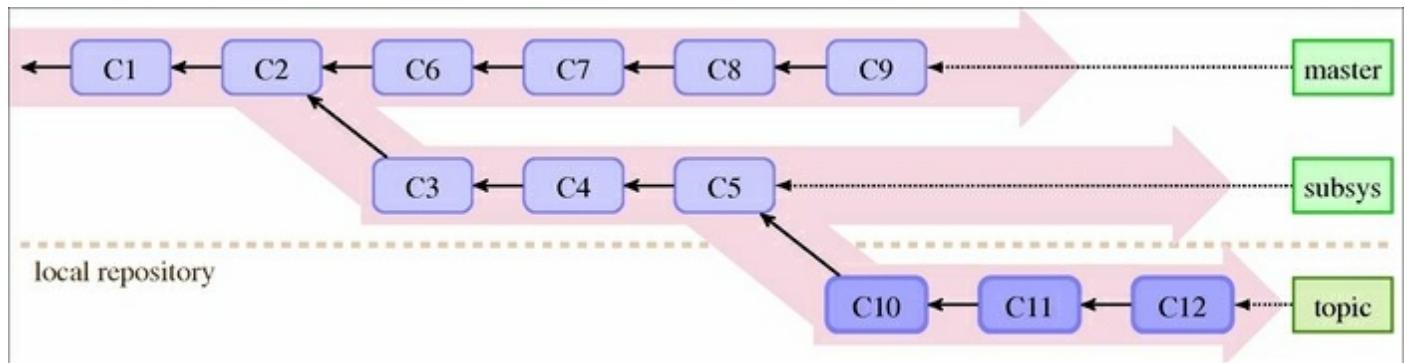


Fig 1: The state of the local repository of a downstream developer before the rewrite of the published history with the new local work that was put on a topic branch

Then, upstream rewrites the `subsys` branch to start from the current (topmost) revision in the `master` branch. This operation is called **rebase**, and was described in the previous chapter, [Chapter 7, Merging Changes Together](#). During rewrite, one of the commits was dropped; perhaps the same change was already present in `master` and was skipped, or perhaps it was dropped for some reason or squashed into the previous commit with an interactive rebase (this operation will be described later in the *Interactive rebase* section). The public repository now looks as follows:

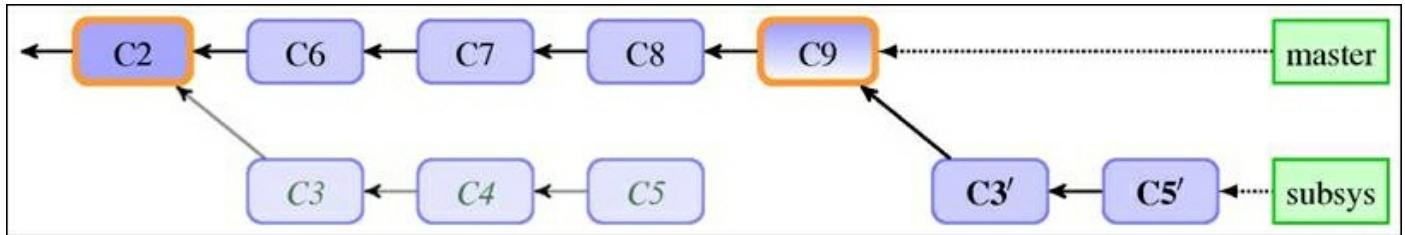


Fig 2: The state of a public upstream repository after rewrite. You can see the emphasized old base of the rebased branch, new base, and rewritten commits (after rebase)

Note that, in the default configuration, Git would refuse to push rewritten history (would deny a nonfast-forward push). You would need to force the push.

The problem is with merging changes based on the pre-rewrite versions of the revisions, such as the `topic` branch in this example:

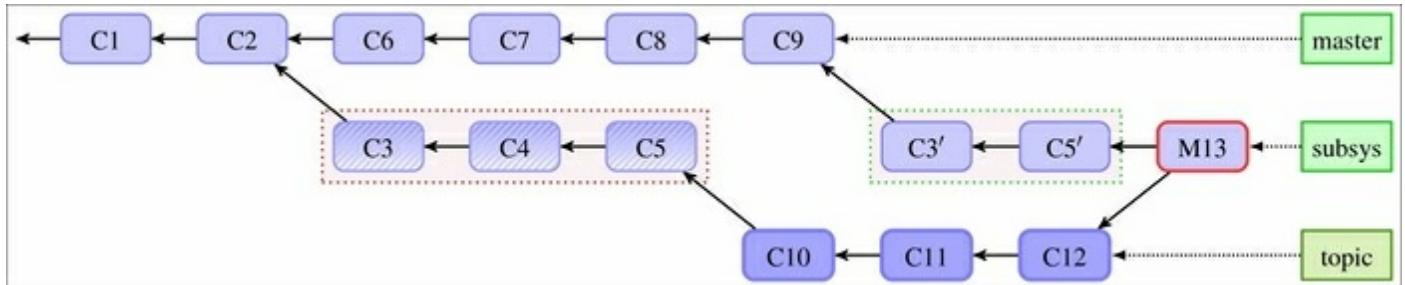


Fig 3: The situation after merging the changes that were based on pre-rewrite revisions into post-rewrite branches. Notice that the merge brings the pre-rewrite version of the revisions, including the commits dropped during rebase

If neither the downstream developer, nor the upstream one, notices that the published history has been rewritten, and merges the changes from the `topic` branch into, for example, the `subsys` branch it was based on, the merge would bring duplicated commits. As we can see in the example in *Fig 3*, after such a merge (denoted by `M13` here), we have both the `C3`, `C4`, and `C5` pre-rewrite commits brought by the `topic` branch,

and the `C3'` and `C5'` post-rewrite commits. Note that the commit `C4` that was removed in the rewrite is back; it might have been a security bug!

Recovering from an upstream history rewrite

But what can we do if the upstream has rewritten the published history (for example, rebased it)? Can we avoid bringing the abandoned commits back, and merging a duplicate or near-duplicate of the rewritten revisions? After all, if the rewrite is published, changing it would be yet another rewrite.

The solution is to rebase your work to fit with the new version from the upstream, moving it from the pre-rewrite upstream revisions to the post-rewrite ones.

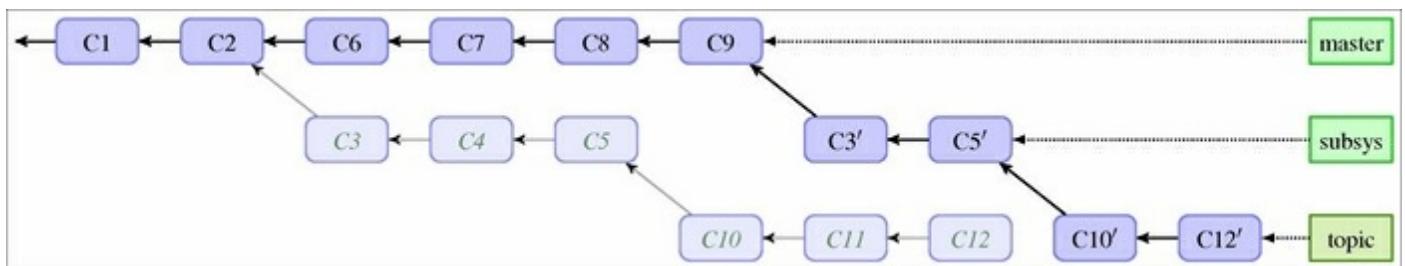


Fig 4: After a downstream rebase of a topic branch, done to recover from upstream rewrite

In the case of our example, it would mean rebasing the `topic` branch onto a new (post-rewrite) version of `subsys`, as shown in Fig 4.

Note

You might not have a local copy of the `subsys` branch; in this case, substitute `subsys` with the respective remote-tracking branch, for example, `origin/subsys`.

Depending on whether the `topic` branch is public or not, it might mean that now you are breaking the promise of unaltered public history for your downstream. Recovering from an upstream rewrite might then result in a ripple of rebases following the rewrite down the river of

downstreams (of dependent repositories).

An easy case is when `subsys` is simply rebased, and the changes remain the same (which means that `c4` vanished because one of `c6-c9` included it). Then, you can simply rebase `topic` on top of its upstream, that is, `subsys`, with:

```
$ git rebase subsys topic
```

The `topic` part is not necessary if you are currently on it (if `topic` is the current branch). This rebases everything: the old version of `subsys` and your commits in `topic`. This solution, however, relies on the fact that `git rebase` would skip repeated commits (removing `c3`, `c4`, and `c5`, and leaving only `c10'` and `c12'`). It might be better and less error-prone to assume the more difficult case.

The hard case is when rewriting `subsys` involved some changes and was not only a pure rebase, or when an interactive rebase was used. In this case, it is better to explicitly move just your changes, namely `subsys@{1}..topic` (assuming that the `subsys@{1}` entry in `subsys` reflog is before rewrite), stating that they are moved on top of new `subsys`. This can be done with the `--onto` option:

```
$ git rebase --onto subsys subsys@{1} topic
```

You can make Git use reflog to find a better common ancestor with the `--fork-point` option to Git rebase, for example:

```
$ git rebase --fork-point subsys topic
```

The rebase would then move the changes to `topic`, starting with the result of the `git merge-base --fork-point subsys topic` command; though if the reflog of the `subsys` branch does not contain necessary information, Git would fall back to upstream; here `subsys`.

Note

Note that you can use an interactive rebase instead of an ordinary rebase like in the narration mentioned earlier, for a better control at the

cost of more work (for example, to drop commits that are already present, but are not detected by the rebase machinery as such).

Amending history without rewriting

What to do if what you need to fix is in the published part of the history? As described in *Perils of rewriting published history section*, changing the parts of the history that were made public (which is actually creating a changed copy and replacing references) can cause problems for downstream developers. You better not to touch this part of the graph of revisions.

There are a few solutions to this problem. The most commonly used is to put a new fixup commit with appropriate changes (for example, a typo fix in a documentation). If you need to remove changes, deciding that they turned out to be bad to have, you can create a commit to revert the changes.

If you fix a commit or revert one, it would be nice to annotate that commit with the information that it was buggy, and which commit fixed (or reverted) it. Even though you cannot (should not) edit the fixed commit to add this information if the commit is public, Git provides a notes mechanism to append extra information to existing commits, which is a bit like publishing an addendum, errata, or amendment. You need however to remember that notes are not published by default, nonetheless it is easy to publish them too (you just need to remember to do it).

Reverting a commit

If you need to back-out an existing commit, undoing the changes it brought, you can use `git revert`. As described in [Chapter 7, Merging Changes Together](#) (see, for example, *Fig 4*), the *revert* operation creates a commit with reverse of changes. For example, where original commit adds a line, reversion removes it, where original commit removes a line, reversion adds it.

Note

Note that different version control systems use the name *revert* for different operations. In particular, it is often used to mean resetting the changes to a file back to latest committed version, throwing away uncommitted changes. It is something that `git reset -- <file>` does in Git.

It is best shown on an example. Let's take for example of the last commit on branch `multiple`, and check the summary of its changes:

```
$ git show --stat multiple
commit bb71a804f9686c4bada861b3fc3cfb5600d2a47
Author: Alice Developer <alice@company.com>
Date:   Sun Jun 1 03:02:09 2014 +0200

        Support optional <count> parameter

src/rand.c | 26 ++++++-----+
1 file changed, 21 insertions(+), 5 deletions(-)
```

Reverting this commit (which requires a clean working directory) would create a new revision. This revision undoes the changes that the reverted commit brought:

```
$ git revert bb71a80
[master 76d9e25] Revert "Support optional <count> parameter"
1 file changed, 5 insertions(+), 21 deletions(-)
```

Git would ask for a commit message, which should explain why you reverted a revision, how it was faulty, and why it needed to be reverted rather than fixed. The default is to give the SHA-1 of the reverted commit:

```
$ git show --stat
commit 76d9e259db23d67982c50ec3e6f371db3ec9efc2
Author: Alice Developer <alice@example.com>
Date:   Tue Jun 16 02:33:54 2015 +0200
Revert "Support optional <count> parameter"
```

This reverts commit bb71a804f9686c4bada861b3fc3cfb5600d2a47.

```
src/rand.c | 26 ++++++-----  
1 file changed, 5 insertions(+), 21 deletions(-)
```

An often found practice is to leave alone the subject (which allows to easily find reverts), but replace the content with a description of the reasoning behind the revert.

Reverting a faulty merge

Sometimes, you might need to undo an effect of a merge. Suppose that you have merged changes, but it turned out that they were merged prematurely, and that the merge brings regressions.

Let's say that the branch that got merged is a `topic` branch and that you were merging it into the `master` branch. This situation is shown in *Fig 5*:

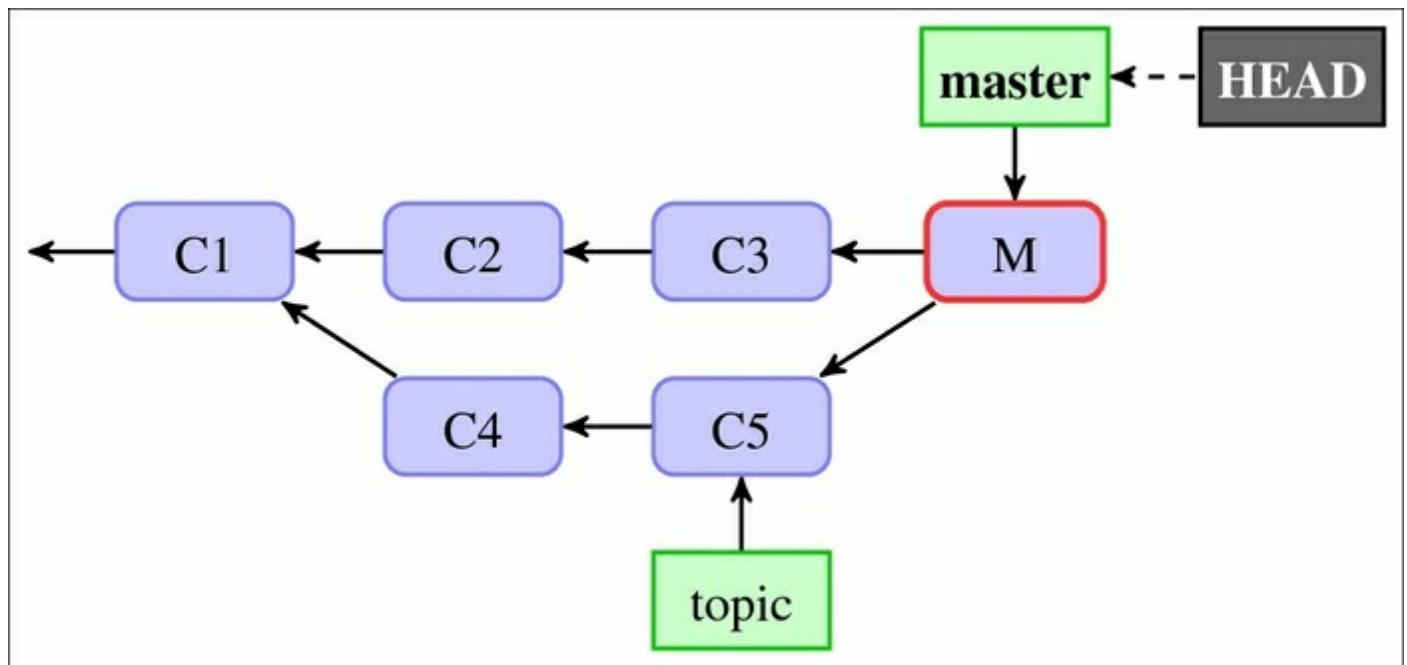


Fig 5: An accidental or premature merge commit, a starting point to reverting merges and redoing reverted merges

If you didn't publish this merge commit before you noticed the mistake and the unwanted merge exists only in your local repository, the easiest solution is to drop this commit with `git reset --hard HEAD^` (see [Chapter 4, Managing Your Worktree](#) for an explanation of the hard

mode of `git reset`).

What do you do if you realize only later that the merge was incorrect, for example, after one more commit was created on the `master` branch and published? One possibility is to revert the merge.

However, a merge commit has more than one parent, which means more than one delta (more than one changeset). To run revert on a merge commit, you need to specify which patch you are reverting or, in other words, which parent is the mainline. In this particular scenario, assuming that there was one more commit after the merge (and that the merge was two commits back), the command would look as follows:

```
$ git revert -m 1 HEAD^^  
[master b2d820c] Revert "Merge branch 'topic'"
```

The situation after reverting a merge is shown in *Fig 6*:

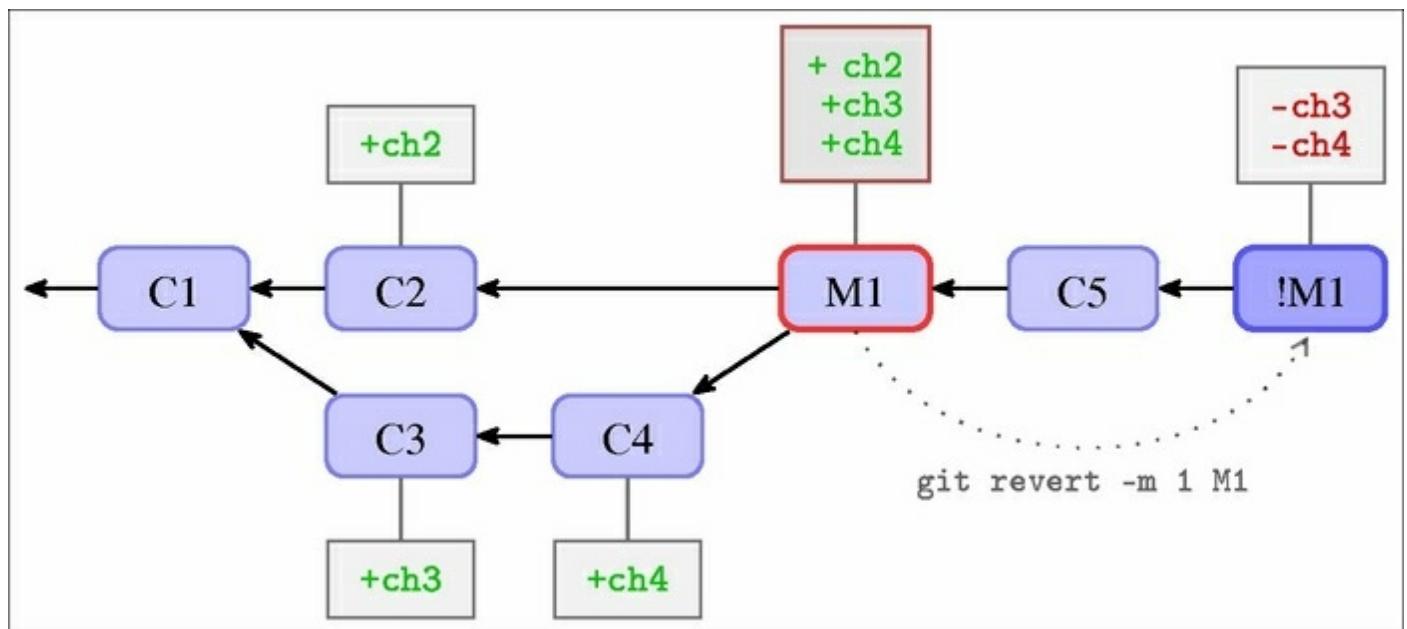


Fig 6: The history from the previous figure after `git revert -m 1 <merge commit>`. The square boxes attached to the selected commits symbolize their changesets in a diff-like format (combined diff format for the merged commit)

Starting with the new `!M1` commit (the symbol `!M1` was used to symbolize

negation or reversal of commit M1), it's as if the merge never happened, at least, with respect to the changes.

Recovering from reverted merges

Let's assume that you continued work on a branch whose merge was reverted. Perhaps it was prematurely merged, but it doesn't mean that the development on it stopped. If you continue to work on the same branch, perhaps by creating commits with fixes, they will get ready in some time and then you will need to be able to merge them correctly into the mainline, again. Or perhaps, the mainline would mature enough to be able to accept a merge. Trouble lies ahead if you simply try to merge your branch again, the same way as the last time.

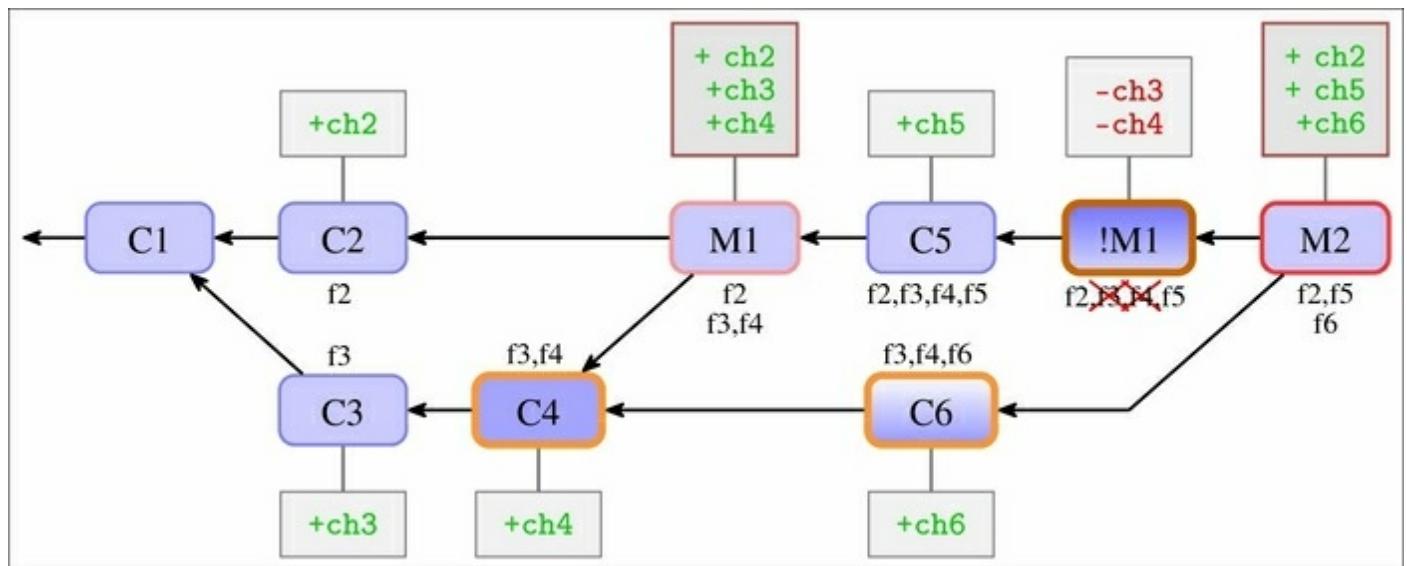


Fig 7: The unexpectedly erroneous result of trying to simply redo reverted merges in a history with a bad merge. The text beside the commits represents a list of features present in or absent from a commit. The three commits with a thick outline are merged commits ("ours" and "theirs" version) and the merge base: the common ancestor ("base")

The unexpected result is that Git has brought only the changes since the reverted merge. The changes brought by the commits on a side branch whose merge got reverted are not here. In other words, you would get a

strange result: the new merge would not include the changes that were created on your branch (on side branch) before the merge that got reverted.

This is caused by the fact that revert undoes changes (the data), but does not undo the history (the DAG of revisions). This means that a new merge sees `c4`, the commit on the side branch just before the reverted merge, as a common ancestor. Because the default three-way merge strategy looks only at the state of the *ours*, *theirs*, and base snapshot, it doesn't search through the history to find that there was a revert there. It sees that both the common ancestor `c4` and the merged branch (that is, *theirs*) `c6` do include features brought by the commits `c3` and `c4`, namely `f3` and `f4`, while the branch that we merged into (that is, *ours*) doesn't have them because of the revert.

For the merge strategy, it looks exactly like the case where one branch deleted something, which means that this change (this removal) is the result of the merge (looks like the case when there was change only in one side). Particularly, it looks like the base has a feature, the side branch has a feature, but the current branch doesn't (because of the revert), so the result doesn't have it. You can find the explanation of the merging mechanism in [Chapter 7, Merging Changes Together](#).

There is more than one option to fix this issue and make Git re-merge the `topic` branch correctly, which means including features `f3` and `f4` in the result. Which option you should choose depends on the exact circumstances, for example, whether the branch being merged is published or not. You don't usually publish topic branches, and if you do, perhaps in the form of the `proposed-updates` branch with all the topic branches merged in, it is with the understanding that they can and probably will be rewritten.

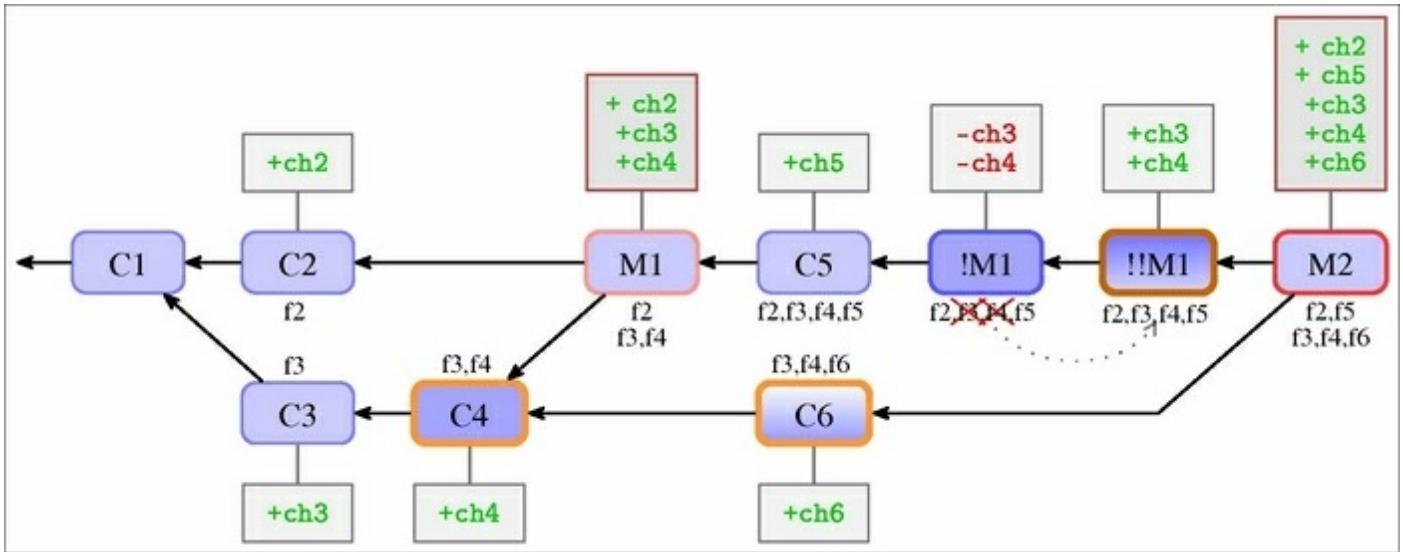


Fig 8: The history after remerging (as M_2) a reverted merge M_1 by reverting the revert $!!M_1$. Notation used like in Fig 7

One option is to bring back deleted changes by reverting the revert. The result is shown in *Fig 8*. In this case, you have brought changes to match the recorded history.

Another option would be to change the view of the history (perhaps temporarily), for example amending it with `git replace`, by changing the merge $!M_1$ to a nonmerge commit. Both these options are suitable in the situation where at least the parts of the branch being merged, namely `topic`, were published.

If the problem was some bugs in the commits being merged (on the branch `topic`) and the branch being merged was not published, you can fix these commits with the interactive rebase, as described earlier.

Rebasing changes the history anyway, so if you additionally ensure that the new history you are creating with the rebase does not have any revision in common with the old history that includes the said failed and reverted merge, re-merging the topic branch would pose no challenges.

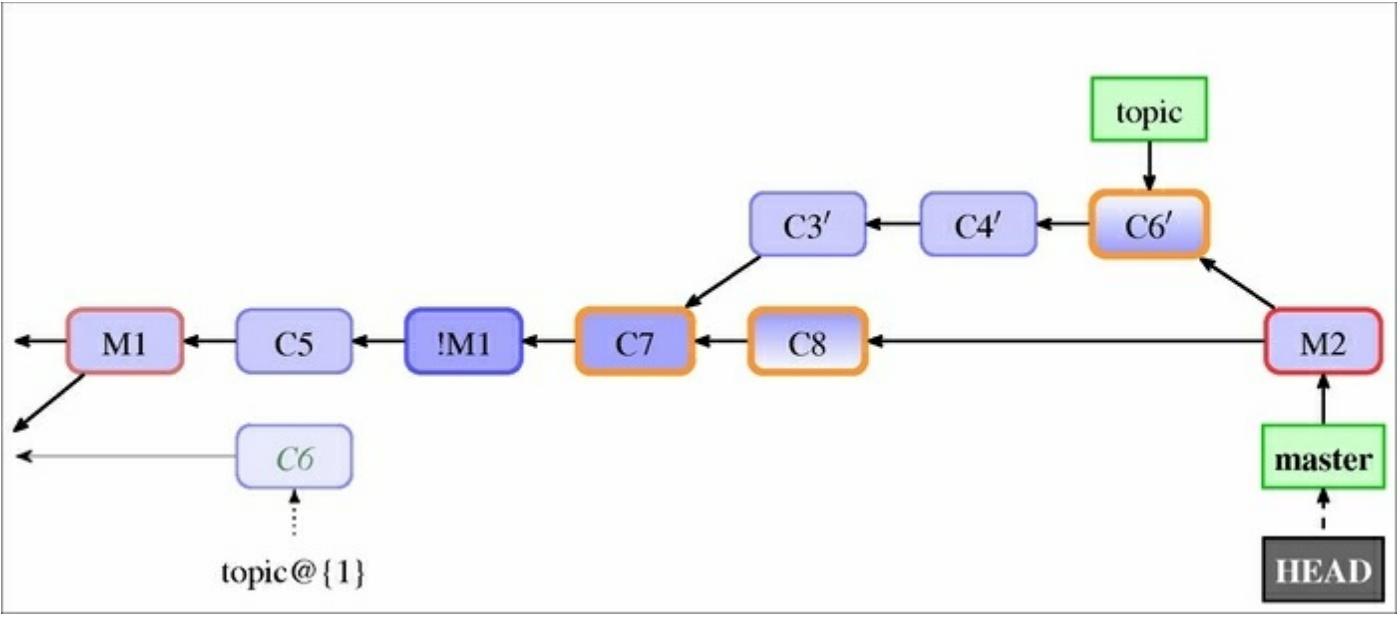


Fig 9: The history after remerging the rebased branch, which had its merge reverted. The rest of the history that is not visible here is like in Fig 6. The three commits with a thick outline are merged commits (the "ours" and "theirs" version) and the new merge base is the common ancestor ("base")

Usually, you would rebase a topic branch, `topic` here, on top of the current state of the branch it was forked from, which here is the `master` branch. This way, your changes are kept up to date with the current work, which makes a later merge easier. Now that the `topic` branch has new history, merging it again into `master`, like in *Fig 9*, is easy and it doesn't give any surprises or troubles.

A more difficult case would be if the `topic` branch is for some reason (like being able to merge it into the `maint` branch too) required to keep its base. Not more difficult in the sense that there would be problems with re-merging the `topic` branch after rebase, but that we need to ensure that the branch after rebase doesn't share history with the reverted merge arc. The goal is to have history in a shape as shown in *Fig 10*. By default, rebase tries to fast-forward revisions if they didn't change (for example, leaving `c3` in place if the rebase didn't modify it), so we need to use `-f` / `--force-rebase` to force rebasing also of

unchanged skippable commits. (or `--no-ff`, which is equivalent).

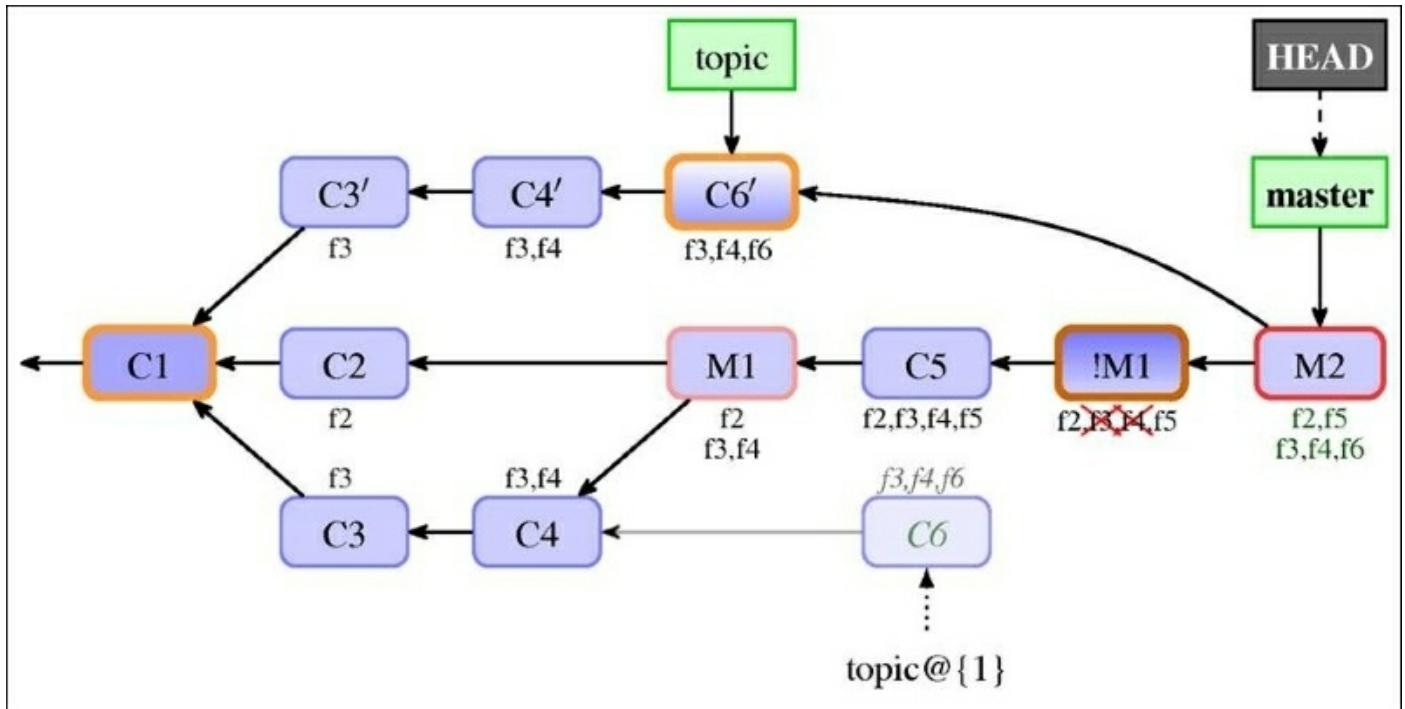


Fig 10: The history after remerging an "in place" rebased topic branch, where a pre-rebase merge was reverted. The notations used to mark the commits are the same as in Fig 7

So, you should not be blindly reverting the revert of a merge. What to do with the problem of remerging after reverted merge depends on how you want to handle the branch being merged. If the branch is being rewritten (for example, using interactive rebase), then reverting the revert would be actively a wrong thing to do, because you could bring back errors fixed in the rewrite.

Storing additional information with notes

The notes mechanism is a way to store additional information for an object, usually a commit, without touching the objects themselves. You can think of it as an attachment, or an appendix, "*stapled*" to an object. Each note belongs to some category of notes, so that notes used for different purposes can be kept separate.

Adding notes to a commit

Sometimes, you want to add extra information to a commit—an information that is available only after its creation. It might be, for example, a note that there was a bug found in the said commit, and perhaps, even that it got fixed in some specified future commit (in case of regression). Perhaps, we realized, after the commit got published, that we forgot to add some important information to the commit message, for example, explain why it was done. Or maybe, we realized that there is another way of doing it and we want to note it to not forget about it, and for other developers to share the idea.

Because in Git history is immutable; you cannot do this without rewriting the history (creating a modified copy and forgetting the old version of the history). Immutability of history is important: it allows people to sign revisions and trust that, once inspected, history cannot change. What you can do instead is to add the extra message as a note.

Let's assume that codevelopers have switched from `atoi()` to `strtol()`, because the former is deprecated. The change was since then made public. But the commit message didn't include an explanation of why it was deprecated and why it is worth it to switch, even if the code after the change is longer. Let's add the information as a note:

```
$ git notes add \
-m 'atoi() invokes undefined behaviour upon error' v0.2~3
```

We have added the note directly from the command line without invoking the editor by using the `-m` flag (the same as for `git commit`) to simplify the explanation of this example. The note will be visible while running `git log` or `git show`:

```
$ git show --no-patch v0.2~3
commit 8c4ceca59d7402fb24a672c624b7ad816cf04e08
Author: Bob Hacker <bob@company.com>
Date:   Sun Jun 1 01:46:19 2014 +0200
```

Use `strtol()`, `atoi()` is deprecated

Notes:

`atoi()` invokes undefined behaviour upon error

As you can see from the preceding output, our note is shown after the commit message in the `Notes:` section. Displaying notes can be disabled with the `--no-notes` option and (re)enabled with `--show-notes`.

How notes are stored

In Git, notes are stored using extra references in the `refs/notes/` namespace. By default, commit notes are stored using the `refs/notes/commits` ref; this can be changed using the `core.notesRef` configuration variable, which in turn can be overridden with the `GIT_NOTES_REF` environment variable.

The value of either variable must be fully qualified (that is, it must include the `refs/notes/` prefix, though this requirement got relaxed in newest Git). If the given ref does not exist, it is not an error, but it means that no notes should be printed. These variables decide both which type of notes are displayed with the commit after the `Notes:` line, and where to write the note created with `git notes add`.

You can see that the new type of reference has appeared in the repository:

```
$ git show-ref --abbrev
2b953b4 refs/heads/bar
5d25848 refs/heads/master
bb71a80 refs/heads/multiple
fcac4a6 refs/notes/commits
5d25848 refs/remotes/origin/HEAD
5d25848 refs/remotes/origin/master
b35871a refs/stash
995a30b refs/tags/v0.1
ee2d7a2 refs/tags/v0.2
```

If you examine the new reference, you will see that each note is stored in a file named after the SHA-1 identifier of the annotated object. This means that you can have only one note of the given type for one object. You can always edit the note, append to it (with `git notes append`), or replace its content (with `git notes add --force`). In the interactive mode, Git opens the editor with the contents of the note, so edit/append/replace is the same here. As opposed to commits, notes are

mutable:

```
$ git show refs/notes/commits
commit fcac4a649d2458ba8417a6bbb845da4000bbfa10
Author: Alice Developer <alice@example.com>
Date:   Tue Jun 16 19:48:37 2015 +0200

Notes added by 'git notes add'

diff --git a/8c4ceca59d7402fb24a672c624b7ad816cf04e08
b/8c4ceca59d7402fb24a672c624b7ad816cf04e08
new file mode 100644
index 0000000..a033550
--- /dev/null
+++ b/8c4ceca59d7402fb24a672c624b7ad816cf04e08
@@ -0,0 +1 @@
+atoi() invokes undefined behaviour upon error

$ git log -1 --oneline 8c4ceca59d7402fb24a672c624b7ad816cf04e08
8c4ceca Use strtol(), atoi() is deprecated
```

Notes for commits are stored in a separate line of (meta-)history, but this need not be the case for the other categories of notes: the notes reference can point directly to the *tree* object instead of to the *commit* object such as for `refs/notes/commits`.

One important issue that is often overlooked in books and articles is that it is the full path to file with notes contents, not the base name of the file, that identifies the object the note is attached to. If there are many notes, Git can and would use a fan-out directory hierarchy, for example storing the preceding note at the
8c4ceca59d7402fb24a672c624b7ad816cf04e08 path (notice the slashes).

Other categories and uses of notes

Notes are usually added to commits. But even for those notes that are attached to commits it makes sense, at least in some cases, to store different pieces of information using different categories of notes. This makes it possible to decide on an individual basis which parts of information to display, and which parts to push to the public repository,

and it allows to query for specific parts of information individually.

To create a note in a namespace (category) different from the default one (where the default means notes/commits, or core.notesRef if set), you need to specify the category of notes while adding it:

```
$ git notes --ref=issues add -m '#2' v0.2~3
```

Now, by default, Git would display only the core.notesRef category of notes after the commit message. To include other types of notes, you must either select the category to display with `git log --notes=<category>` (where <category> is either the unqualified or qualified reference name, or a glob; you can use `--notes=*` to show all the categories), or configure which notes to display in addition to the default with the `display.notesRef` configuration variable (or the `GIT_NOTES_DISPLAY_REF` environment variable). You can either specify the configuration variable value multiple times, just like for `remote.<remote-name>.push` (or specify a colon-separated list of pathnames in the case of using an environment variable), or you can specify a globbing pattern:

```
$ git config notes.displayRef 'refs/notes/*'  
$ git log -1 v0.2~3  
commit 8c4ceca59d7402fb24a672c624b7ad816cf04e08  
Author: Bob Hacker <bob@company.com>  
Date:   Sun Jun 1 01:46:19 2014 +0200
```

Use `strtol()`, `atoi()` is deprecated

Notes:

`atoi()` invokes undefined behaviour upon error

Notes (issues):

#2

There are many possible uses of notes. You can, for example, use notes to reliably mark which patches (which commits) were *upstreamed* (forward-ported to the development branch) or *downstreamed* (backward-ported to the more stable branch or to the stable repository), even if the upstreamed/downstreamed version is not identical, and mark a patch as

being *deferred* if it is not ready for either upstream or downstream.

This is a bit more reliable, if requiring manual input, than relying on the mechanism of `git patch-id` to detect when changeset is already present (which you can use by rebasing, using `git cherry`, or with the `--cherry` / `--cherry-pick` / `--cherry-mark` option to `git log`). This is, of course, in case we are not using topic branches from the start, but rather we are cherry-picking commits.

Notes can be used to store results of the post-commit (but pre-merge) code audit, and to notify other developers why this version of the patch was used.

Notes can also be used to handle **marking bugs and bug fixes**, and **verifying fixes**. You often find bugs in commits long after they got published, that's why you need notes for this; if you find a bug before publishing, you would rewrite the commit instead.

In this case, first, when the bug gets reported, and if it was regression, you find which revision introduced the bug (for example with `git bisect`, as described in [Chapter 2, Exploring Project History](#)). Then you would want to mark this commit, putting the identifier of a bug entry in an issue tracker for the project (usually, a number or number preceded by some specific prefix such as `Bug:1385`) in the `bugs`, or `defects`, or `issues` category of notes; perhaps you would want to also include the description of a bug. If the bug affects security, it might be assigned a vulnerability identifier, for example, a Common Vulnerabilities and Exposures (CVE) number; this information could be put into the note in the `CVE-IDs` category.

Then, after some time, hopefully, the bug will get fixed. Just like we marked the commit that it contains the bug, we can annotate it additionally with the information on which commit fixes it, for example, in note under `refs/notes/fixes`. Unfortunately, it might happen that the first attempt at fixing it didn't handle the bug entirely correct, and you have to amend a fix, or perhaps even create a fix for a fix. If you are using `bugfix` or `hotfix` branches (topic branches for bugfixes), as

described in [Chapter 6](#), *Advanced Branching Techniques*, it will be easy to find them together and to apply them together—by merging said bugfix branch. If you are not well, then it would be a good idea to use notes to annotate fixes that should be cherry-picked together with a supplementary commit, for example by adding note in `alsoCherryPick`, or `seeAlso`, or whatever you want to name this category of notes. Perhaps also an original submitter, or a Q&A group, would get to the fix and test that it works correctly; it would be better if the commit was tested before publishing, but it is not always possible, so `refs/notes/tests` it is.

Third-party tools use (or could use) notes to store additional **per-commit tool-specific information**. For example, Gerrit, which is a free, web-based team code collaboration tool, stores information about code reviews in `refs/notes/reviews`: including the name and e-mail address of the Gerrit user that submitted the change, the time the commit was submitted, the URL to the change review in the Gerrit instance, review labels and scores (including the identity of the reviewer), the name of project and branch, and so on:

Notes (review) :

```
Code-Review+2: John Reviewer <john@company.com>
Verified+1: Jenkins
Submitted-by: Bob Developer <bob@company.com>
Submitted-at: Thu, 20 Oct 2014 20:11:16 +0100
Reviewed-on: http://localhost:9080/7
Project: common/random
Branch: refs/heads/master
```

Similarly, `git svn`, a tool for bidirectional operation between the Subversion repository and Git working as a fat client for Subversion (`svn`), could have stored the original Subversion identifiers in notes, rather than appending this information to a commit message (or dropping it altogether).

Going to a more exotic example, you can use the notes mechanism to store the result of a build (either the archive, the installation package, or just the executable), attaching it to a commit or a tag. Theoretically, you could store a build result in a tag, but you usually expect for a tag to

contain **Pretty Good Privacy (PGP)** signature and perhaps also the release highlights. Also, you would in almost all the cases want to fetch all the tags, while not everyone wants to pay the cost of disk space for the convenience of pre-build executables. You can select from case to case whether you want or not to fetch the given category of notes (for example, to skip pre-built binaries), while you autofollow tags. That's why notes are better than tags for this purpose.

Here the trouble is to correctly generate a binary note. You can binary-safely create a note with the following trick:

```
# store binary note as a blob object in the repository
$ blob=$(git hash-object -w ./a.out)
# take the given blob object as the note message
$ git notes --ref=built add --allow-empty -C "$blob" HEAD
```

You cannot simply use `-F ./a.out`, as this is not binary safe—comments (or rather what was misdetected as comment, that is lines starting with `#`) would be stripped.

The notes mechanism is also used as a mechanism to enable storing cache for the `textconv` filter (see the section on `gitattributes` in [Chapter 4, Managing Your Worktree](#)). All you need to do is configure the filter, setting its `cachetextconv` to `true`:

```
[diff "jpeg"]
  textconv = exif
  cachetextconv = true
```

Here, notes in the `refs/notes/textconv/jpeg` category (named after the filter) are used to attach the text of the conversion to a blob object.

Rewriting history and notes

Notes are attached to the objects they annotate, usually commits, by their SHA-1 identifier. What happens then with notes when we are rewriting history? In the new, rewritten history, SHA-1 identifiers of objects in most cases are different.

It turns out that you can configure this quite extensively. First, you can

select which categories of notes should be copied along with the annotated object during rewrite with the `notes.rewriteRef` multi-value configuration variable. This setting can be overridden with the `GIT_NOTES_REWRITE_REF` (see the naming convention) environment variable with a colon-separated list (like for the well-known `PATH` environment variable) of fully qualified note references, and globs denoting reference patterns to match. There is no default value for this setting; you must configure this variable to enable rewriting.

Second, you can also configure whether to copy a note during rewriting depending on the exact type of the command doing the rewriting (currently supported are `rebase` and `amend` as the value of the command). This can be done with the Boolean-valued configuration variable `notes.rewrite.<command>`.

In addition, you can decide what to do if the target commit already has a note while copying notes during a rewrite, for example while squashing commits using an interactive rebase. You have to decide between `overwrite` (take the note from the appended commit), `concatenate` (which is the default value), and `ignore` (use the note from the original commit being appended to) for the `notes.rewriteMode` configuration variable, or the `GIT_NOTES_REWRITE_MODE` environment variable.

Publishing and retrieving notes

So, we have notes in our own local repository. What to do if we want to share these notes? How do we make them public? How can we and other developers get notes from other public repositories?

We can employ our knowledge of Git here. Section *How notes are stored* explained that notes are stored in an object database of the repository using special references in the `refs/notes/` namespace. The contents of note are stored as a blob object, referenced through this special ref. Commit notes (notes in `refs/notes/commits`) store the history of notes, though Git allows you to store notes without history as well. So, what you need to do is to get notes references, and the contents of notes will follow. This is the usual mechanism of repository

synchronization (of object transfer).

This means that to **publish your notes**, you need to configure appropriate push lines in the appropriate remote repository configuration (see [Chapter 5, Collaborative Development with Git](#)). Assuming that you are using a separate `public` remote (if you are the maintainer, you will probably use simply `origin`), which is perhaps set as `remote.pushDefault`, and that you would like to publish notes in any category, you can run:

```
$ git config --add remote.public.push  
'+refs/notes/*:refs/notes/*'
```

In the case when `push.default` is set to `matching` (or Git is old enough to have this as the default behavior), or the "push" lines use special refspec ":" or "+:", then it is enough to push notes refs the first time, and they would be pushed automatically each time after:

```
$ git push origin 'refs/notes/*'
```

Fetching notes is only slightly more involved. If you don't produce specified types of notes yourself, you can fetch notes in the mirror-like mode to the ref with the same name:

```
$ git config --add remote.origin.fetch  
'+refs/notes/*:refs/notes/*'
```

However, if there is a possibility of conflict, you would need to fetch notes from the remote into the remote-tracking notes reference, and then use `git notes merge` to join them into your notes; see the documentation for details.

Note

If you wanted to make it easy to merge `git notes`, perhaps even automatically, then following the convention of the `key: value` entries on separate lines for the content of notes with the duplicates removed would help.

There is no standard naming convention for remote-tracking notes references, but you can use either `refs/notes/origin/*` (so that the shortened notes category `commits` from the remote `origin` is `origin/commits` and so on), or go whole works and fetch `refs/*` from the remote origin into `refs/remotes/origin/refs/*` (so the `commits` category would land in `refs/remotes/origin/refs/notes/commits`).

Using the replacements mechanism

The original idea for the replace-like/replacement-like mechanism was to make it possible to join the history of two different repositories.

The original impulse was to be able to switch from the other version control system to Git by creating two repositories: one for the current work, starting with the most recent version in the empty repository, and the second one for the historical data, storing the conversion from the original system. That way, it would be possible to take time doing the faithful conversion of historical data, and even fix it if the conversion were incorrect, without affecting the current work.

What was needed is some mechanism to connect histories of those two repositories, to have full history for inspection going back to the creation of a project (for example, for `git blame`, that is, the line-history annotation).

The replacements mechanism

The modern incarnation of such tools is a replace (or replacements) mechanism. With it, you can replace any object, with any object or rather create a virtual history (virtual object database of a repository) by creating an overlay so that most Git commands return a replacement in place of the original object.

But the original object is still there, and Git's behavior with respect to the replacement mechanism was done in such a way as to eliminate the possibility of losing data. You can get the original view with the `--no-replace-objects` option to the `git` wrapper before the command, or the

`GIT_NO_REPLACE_OBJECTS` environment variable. For example, to view the original history, you can use `git --no-replace-objects log`.

The information about replacement is saved in the repository by storing the ref named after SHA-1 of the replaced object in the `refs/replace/` namespace, with the SHA-1 of replacement as its sole content. However, there is no need to edit it by hand or with the low-level plumbing commands; you can use the `git replace` command.

Almost all the commands use replacements, unless told not to, as explained previously. The exception are the reachability analysis commands; this means that Git would not remove the replaced objects because there are no longer reachable if we take replacement into account. Of course, replacement objects are reachable from the replaced refs.

You can replace any object with any object, though changing the type of an object requires telling Git that you know what you are doing with `git replace -f <object> <replacement>`. This is because such a change might lead to troubles with Git, because it was expecting one type of object and getting another.

With `git replace --edit <object>`, you can edit its contents interactively. What really happens is that Git opens the editor with the object contents and, after editing, Git creates a new object and a replacement ref. The object format (in particular, the commit object format, as one would almost always edit commits) was described at beginning of this chapter. You can change the commit message, commit parents, authorship, and so on.

Example – joining histories with `git replace`

Let's assume that you want to split the repository into two, perhaps for performance reasons. But you want to be able to treat joined history as if it were one. Or perhaps, there was a history split after the SCM change, with the fresh repository with the current work (started after switching from the current state of a project with an empty history) and

the converted historical repository kept separate.

How to split history was described in the examples of using `git filter-branch` here in this chapter. One of solutions shown here was to run `git replace --graft <to be root>` on a commit where you want to split and then use `git filter-branch -- --all` without filters to make the split permanent.

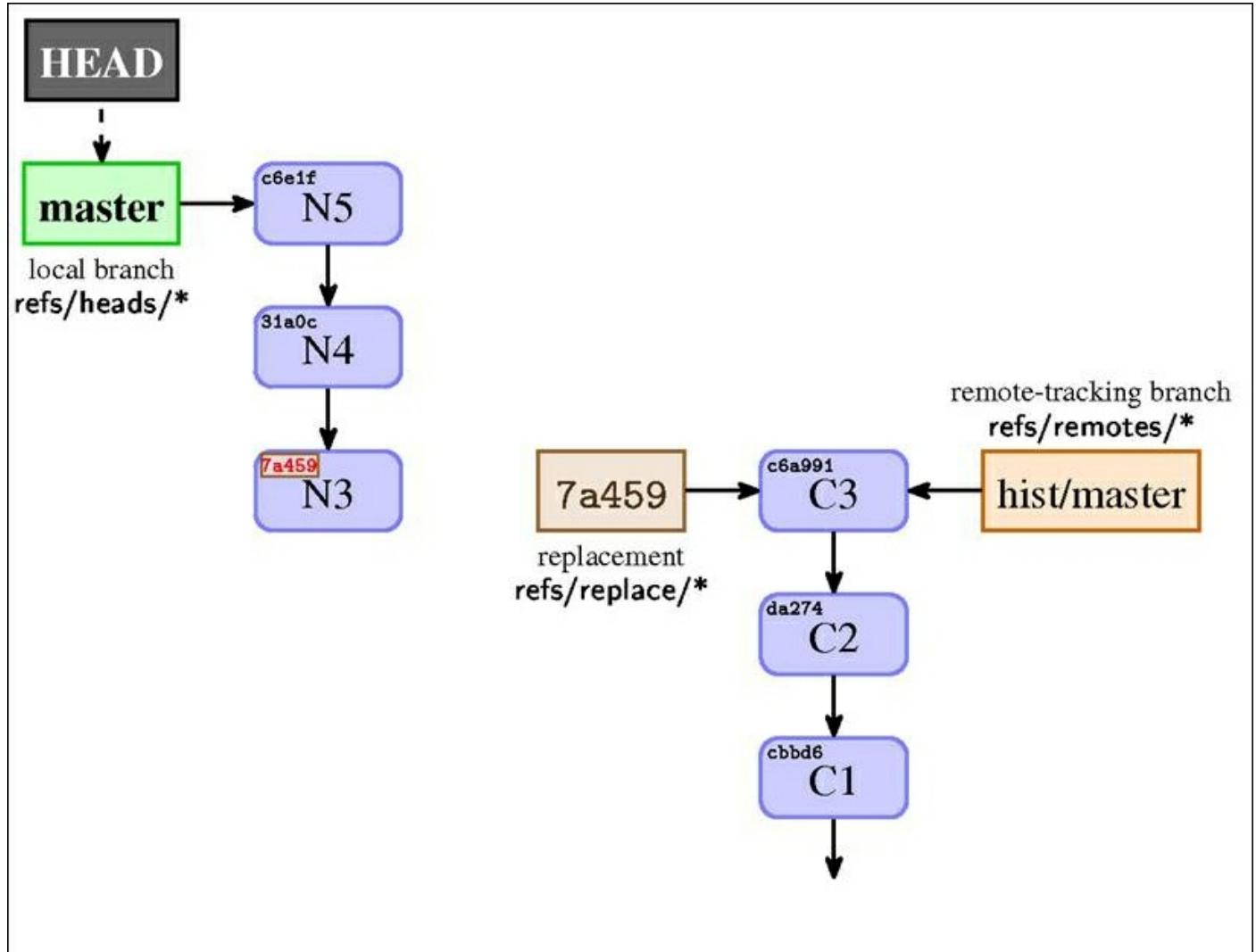


Fig 11: The view of a split history with the replacements turned off (`git --no-replace-objects`). The SHA-1 in the left upper corner of a commit denotes its identifier. Note that SHA-1 identifiers were all shortened to 5 hex-digits in this figure

In many cases, you might want to create a kind of informational commit on top of the historical repository, for example, adding to the `README` file

the notification where one can find the current work repository. Such commits for simplicity are not shown in *Fig 11*.

How to join history depends a bit on whether the history was originally split or was originally joined. If it was originally joined, then split; just tell Git to replace post-split with the pre-split version with `git replace <post-split> <pre-split>`. If the repository was split from beginning, use the `--edit` or `--graft` option to `git replace`.

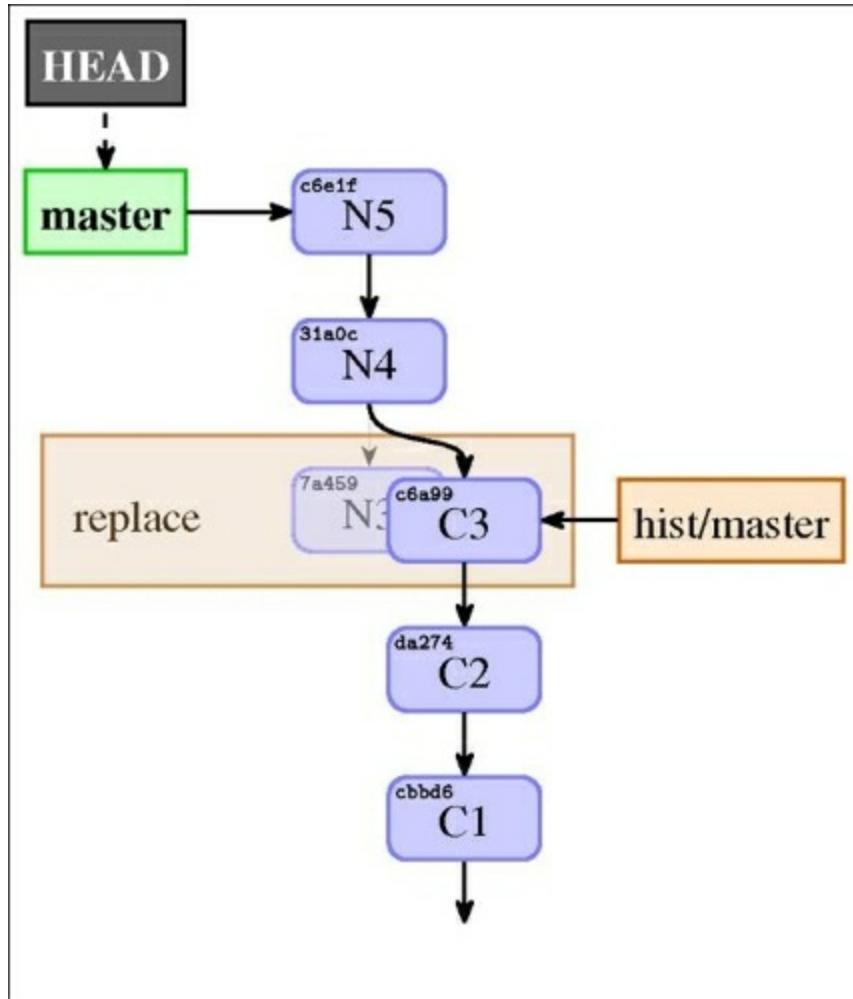


Fig 12: The view of a split history joined using replacements. The notations are the same as in the previous figure, but with the replace ref shown in a different way—as the result of the replacement

The split history is there, just hidden from the view. For all the Git commands, the history looks like *Fig 12*. You can, as described earlier, turn off using replacements; in this case, you would see the history as in

Fig 11.

Historical note – grafts

The first attempt to create a mechanism to make it possible to join lines of history was `grafts`. It is a simple `.git/info/grafts` file with the SHA-1 of the affected commit and its replacement parents in line separated by spaces.

This mechanism was only for commits and allowed only to change the parentage. There was no support for transport, that is, for propagating this information from inside of Git. You could not turn `grafts` mechanism off temporarily, at least not easily. Moreover, it was inherently unsafe, because there were no exceptions for the reachability-checking commands, making it possible for Git to remove needed objects by accident during pruning (garbage collecting).

However, you can find its use in examples. Nowadays, it is obsolete, especially with the existence of the `git replace --graft` option. If you use `grafts`, consider replacing them with the replacements objects; there is the `contrib/convert-grafts-to-replace-refs.sh` script that can help with this in the Git sources.

Note

The **shallow clone** (the result of `git clone --depth=<N>`, a clone with the shortened history) is managed with a graft-like `.git/shallow` file. This file is managed by Git, however, not by the user.

Publishing and retrieving replacements

How to publish replacements and how to get them from the remote repository? Because replacements use references, this is quite simple.

Each replacement is a separate reference in the `refs/replaces/` namespace. Therefore, you can get all the replacements with the globing `fetch` or `push` line:

```
+refs/replace/* :refs/replace/*
```

There can be only one replacement for an object, so there are no problems with merging replacements. You can only choose between one replacement or the other.

Theoretically, you could also request individual replacements by fetching (and pushing) individual replacement references instead of using glob.

Summary

This chapter, along with [Chapter 6](#), *Advanced Branching Techniques*, provided all the tools required to manage a clean, readable, and easy-to-review history of a project.

You have learned how to make history more clean by rewriting it, and what does rewriting history mean in Git, when and why to avoid it, and how to recover from an untimely upstream rewrite. You have learned to use an interactive rebase to delete, reorder, squash, and split commits, and how to test each commit during rebase. You know how to do large-scale scripted rewrite with filter-branch: how to edit commits and commit metadata and how to permanently change history, for example, splitting it in two. You also got to know some third-party external tools, which can help with these tasks.

You learned what to do if you cannot rewrite history: how to fix mistakes by creating commits with appropriate changes (for example with `git revert`), how to add extra information to the existing commits with notes, and how to change the virtual view of the history with replacements. You learned to handle reverting a faulty merge and remerging after reverted merge. You know how to fetch and publish both notes and replacements.

To really understand advanced history rewriting and the mechanism behind notes and replacements, this chapter explained the basics of Git internals and low-level commands usable for scripting (including scripted rewrite).

The following chapter, [Chapter 9](#), *Managing Subprojects - Building a Living Framework*, will explain and show different ways to connect different subprojects in one repository, from submodules to subtrees.

You will also learn techniques to manage or mitigate managing large-size assets inside a repository. Splitting a large project into submodules is one, but not the only way to handle this issue.

Chapter 9. Managing Subprojects – Building a Living Framework

In [Chapter 5, Collaborative Development with Git](#), you have learned how to manage multiple repositories, while [Chapter 6, Advanced Branching Techniques](#), taught us various development techniques utilizing multiple branches and multiple lines of development in these repositories. Up till now, these multiple repositories were all repositories of a single project. Different projects were all being developed independent of each other. Repositories of the different projects were autonomous.

This chapter will explain and show different ways to connect different subprojects in the one single repository of the framework project, from the strong inclusion by embedding the code of one project in the other (subtrees), to the light connection between projects by nesting repositories (submodules). You will learn how to add a subproject to a master project, how to update the superproject state, and how to update a subproject. We will find out how to send our changes upstream, backporting them into the appropriate project , and pushing to appropriate repository. Different techniques of managing subprojects have different advantages and drawbacks here.

Submodules are sometimes used to manage large size assets. This chapter would also present alternate solutions to the problem of handling large binary files, and other large assets in Git.

In this chapter, we will cover the following topics:

- Managing library and framework dependencies
- Dependency management tools—managing dependencies outside Git
- Importing code into a superproject as a subtree
- Using subtree merges; the `git-subtree` and `git-stree` tools
- Nested repositories: a subproject inside a superproject

- Internals of submodules: `gitlinks`, `.gitmodules`, the `.git` file
- Use cases for subtrees and submodules, comparison of approaches
- Alternative third-party solutions and tools/helpers
- Git and large files

Managing library and framework dependencies

There are various reasons to join an external project to your own project. Because there are different reasons to include a project (let's call it a *subproject*, or a *module*) inside another project (let's call it *superproject*, or a *master project*, or a *container*), there are different types of inclusions geared towards different circumstances. They all have their advantages and disadvantages, and it is important to understand these to be able choose the correct solution for your problem.

Let's assume that you work on a web application, and that your webapp uses JavaScript (for example, for AJAX, as single-page app perhaps). To make it easier to develop, you probably use some JavaScript library or a web framework, such as jQuery.

Such a library is a separate project. You would want to be able to pin it to a known working version (to avoid problems where future changes to the library would make it stop working for your project), while also being able to review changes and automatically update it to the new version. Perhaps, you would want to make your own changes to the library, and send the proposed changes to the upstream (of course, you would want for users of your project to be able to use the library with your out-of-tree fixes, even if they are not yet accepted by original developers). Conceivably, you might have customizations and changes that you don't want to publish (send to the upstream), but you might still make them available.

This is all possible in Git. There are two main solutions for including subprojects: importing code into your project with the *subtree* merge

strategy and linking subprojects with *submodules*.

Both submodules and subtrees aim to reuse the code from another project, which usually has its own repository, putting it somewhere inside your own repository's working directory tree. The goal is usually to benefit from the central maintenance of the reused code across a number of container repositories, without having to resort to clumsy, unreliable manual maintenance (usually by copy-pasting).

Sometimes, it is more complicated. The typical situation in many companies is that they use many in-house produced applications, which depend on the common utility library or on a set of libraries. You would usually want to develop each of such applications separately, use it together with others, branch and merge, and apply your own changes and customizations, all in a separate Git repository. Though there are cases for having a single monolithic repository, such as simplified organizations, dependencies, cross-project changes, and tooling if you can get away with it.

But this division, one Git repository for one application, is not without problems. What to do with the common library? Each application uses some specific version of the library and you need to supervise which one. If the library gets improved, you need to test whether this new version correctly works with your code and doesn't crash your application. But the common library is not usually developed as a standalone; its development is driven by the needs of projects that use it. Developers improve it to enhance it with new features needed for their applications. At some point of time, they would want to send their changes to the library itself to share their changes with other developers, if only to share the burden of maintaining these features (the out-of-tree patches bring maintenance costs to keep them current).

What to do then? This chapter describes a few strategies used to manage subprojects. For each technique, we will detail how to add such subprojects to superprojects, how to keep them up to date, how to create your own changes, and how to publish selected changes upstream.

Note

Note that all the solutions require that all the files of a subproject are contained in a single subdirectory of a superproject. No currently available solution allows you to mix the subproject files, with other files or have them occupy more than one directory.

However you manage subprojects, be it subtrees, submodules, third-party tools or dependency management outside Git, you should strive for the module code to remain independent of the particularities of the superproject (or at least, handle such particularities using an external, possibly nonversioned configuration). Using superproject-specific modifications goes against modularization and encapsulation principles, unnecessarily coupling the two projects.

Managing dependencies outside Git

In many cases, the technological context (the development stack used) allows to use for packaging and formal dependency management. If it is possible, it is usually preferable to go this route. It lets you split your codebase better and avoid a number of side effects, complications, and pitfalls that litter the submodule and subtree solution space (with different complications for different techniques). It removes the version control systems from the managing modules. It also lets you benefit from versioning schemes, such as **semantic versioning** (<http://semver.org/>), for your dependencies.

As a reminder, here's a partial list (in the alphabetical order) of the main languages and development stacks, and their dependency management/packaging systems and registries (see the full comparison at <http://www.modulecounts.com/>):

- Clojure has Clojars
- Go has GoDoc
- Haskell has Hackage (registry) and cabal (application)
- Java has Maven Central (Maven and Gradle)
- JavaScript has npm (for Node.js) and Bower

- .NET has NuGet
- Objective-C has CocoaPods
- Perl has CPAN (Comprehensive Perl Archive Network) and carton
- PHP has Composer, Packagist, and good old PEAR and PECL
- Python has PyPI (Python Package Index) and pip
- Ruby has Bundler and RubyGems
- Rust has Crates

Sometimes, these are not enough. You might need to apply some out-of-tree patches (changes) to customize the module (subproject) for your needs. But for some reason, you are unable to publish these changes upstream, to have them accepted. Perhaps, the changes are relevant only to your specific project, or the upstream is slow to respond to the proposed changes, or perhaps there are license considerations. Maybe the subproject in question is a in-house module that cannot be made public and which you are required to use for your company projects.

In all these cases, you need for the custom package registry (the package repository) to be used in addition to the default one, or you need to make subprojects be managed as private packages, which these systems often allow. If there is no support for private packages, a tool to manage the private registry, such as Pinto or CPAN::Mini for Perl, would be also needed.

Manually importing the code into your project

Let's take a look at one of the possibilities: why don't we simply import the library into some subdirectory in our project? If you need to bring it up to date, you would just copy the new version as a new set of files. In this approach, the subproject code is embedded inside the code of the superproject.

The simplest solution would be to just overwrite the contents of the subproject's directory each time we want to update the superproject to use the new version. If the project you want to import doesn't use Git, or

if it doesn't use a version control system at all, or if the repository it uses is not public, this will indeed be the only possible solution.

Tip

Using repositories from a foreign VCS as a remote

If the project you want to import (to embed) uses a version control system other than Git, but there is a good conversion mechanism (for example, with a fast-import stream), you can use **remote helpers** to set up a foreign VCS repository as a remote repository (via automatic conversion). You can check [Chapter 5, Collaborative Development with Git](#), and [Chapter 10, Customizing and Extending Git](#) for more information.

This can be done, for example, with the Mercurial and Bazaar repositories, thanks to the `git-remote-hg` and `git-remote-bzr` helpers.

Moving to the new version of the imported library is quite simple (and the mechanism easy to understand). Remove all the files from the directory, add files from the new version of the library, for example by extracting them from the archive, then use `git add` command to the directory:

```
$ rm -rf mylib/
$ git rm mylib
$ tar -xzf /tmp/mylib-0.5.tar.gz
$ mv mylib-0.5 mylib
$ git add mylib
$ git commit
```

This method works quite well in simple cases with the following caveats:

- In Git, in the history of your project, you have only the versions of the library at the time of imports. On the one hand, this makes your project history clean and easy to understand, on the other hand, you don't have access to the fine-grained history of a subproject. For example, when using `git bisect`, you would be able only find that it was introduced by upgrading the library, but not the exact commit in the history of the library that introduced the bug in question.

- If you want to customize the code of the library, fitting it to your project by adding the changes dependent on your application, you would need to reapply those customization in some way after you import a new version. You could extract your changes with `git diff` (comparing it to the unchanged version at the time of import) and then use `git apply` after upgrading the library. Or, you could use a rebase, an interactive rebase, or some patch management interface; see [Chapter 8, Keeping History Clean](#). Git won't do this automatically.
- Each importing of the new version of the library requires running a specific sequence of commands to update superproject: removing the old version of files, adding new ones, and committing the change. It is not as easy as running `git pull`, though you can use scripts or aliases to help.

A Git subtree for embedding the subproject code

In a slightly more advanced solution, you use the **subtree merge** to join the history of a subproject to the history of a superproject. This is only somewhat more complicated than an ordinary pull (at least, after the subproject is imported), but provides a way to automatically merge changes together.

Depending on your requirements, this method might fit well with your needs. It has the following advantages:

- You would always have the correct version of the library, never using the wrong library version by an accident
- The method is simple to explain and understand, using only the standard (and well-known) Git features. As you will see, the most important and most commonly used operations are easy to do and easy to understand, and it is hard to go wrong.
- The repository of your application is always self-contained; therefore, cloning it (with plain old `git clone`) will always include everything that's needed. This means that this method is a good fit for the *required dependencies*.

- It is easy to apply patches (for example, customizations) to the library inside your repository, even if you don't have the commit rights to the upstream repository.
- Creating a new branch in your application also creates a new branch for the library; it is the same for switching branches. That's the behavior you expect. This is contrasted with the submodule's behavior (the other technique for managing subprojects).
- If you are using the `subtree` merge strategy (described shortly in [Chapter 7, Merging Changes Together](#)), for example with `git pull -s subtree`, then getting a new library version will be as easy as updating all the other parts of your project.

Unfortunately however, this technique is not without its disadvantages. For many people and for many projects, these disadvantages do not matter. The simplicity of the subtree-based method usually prevails over its faults.

Here are the problems with the subtree approach:

- Each application using the library doubles its files. There is no easy and safe way to share its objects among different projects and different repositories. Though see the following about the possibility of sharing Git object database.
- Each application using the library has its files checked out in the working area, though you can change it with the help of the **sparse checkout** (described later in the chapter).
- If your application introduces changes to its copy of the library, it is not that easy to publish these changes and send them upstream. Third-party tools such as `git subtree` or `git stree` can help here. They have specialized subcommands to extract the subproject's changes.
- Because of the lack of separation between the subproject files and the superproject files, it is quite easy to mix the changes to the library and the changes to the application in one commit. In such cases, you might need to rewrite the history (or the copy of a history), as described in [Chapter 8, Keeping History Clean](#).

The first two issues mean that subtrees are not a good fit to manage the subprojects that are *optional dependencies* (needed only for some extra features) or *optional components* (such as themes, extensions, or plugins), especially those that are installed by a mere presence in the appropriate place in the filesystem hierarchy.

Tip

Sharing objects between forks (copies) with alternates

You can mitigate the duplication of objects in the repository with alternates or, in other words, with `git clone --reference`. However, then you would need to take greater care about garbage collection. The problematic parts are those parts of the history that are referenced in the borrower repository (that is, one with alternates set up), but are not referenced in the lender reference's repository. The description and explanation of the alternative mechanisms will be presented in [Chapter 11, Git Administration](#).

There are different technical ways to handle and manage the subtree-imported subprojects. You can use classic Git commands, just using the appropriate options while affecting the subproject, such as `--strategy=subtree` (or the `subtree` option to the default `recursive` merge strategy, `--strategy-option=subtree=<path>`) for `merge`, `cherry-pick`, and related operations. This manual approach works everywhere, is actually quite simple in most cases, and offers the best degree of control over operations. It requires, however, a good understanding of the underlying concepts.

In modern Git (since version 1.7.11), there is the `git subtree` command available among installed binaries. It comes from the `contrib/` area and is not fully integrated (for example, with respect to its documentation). This script is well tested and robust, but some of its notions are rather peculiar or confusing, and this command does not support the whole range of possible subtree operations. Additionally, this tool supports only the *import with history* workflow (which will be defined later), which some say clutters the history graph.

There are also other third-party scripts that help with subtrees; among them is `git-stree`.

Creating a remote for a subproject

Usually, while importing a subproject, you would want to be able to update the embedded files easily. You would want to continue interacting with the subproject. For this, you would add that subproject (for example, the common library) as a remote reference in your own (super) project and fetch it:

```
$ git remote add mylib_repo https://git.example.com/mylib.git
$ git fetch mylib_repo
warning: no common commits
remote: Counting objects: 12, done.
remote: Total 12 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (12/12), done.
From https://git.example.com/mylib.git
 * [new branch]      master      -> mylib_repo/master
```

You can then examine the `mylib_repo/master` remote-tracking branch, which can be done either by checking it out into the detached HEAD with `git checkout mylib_repo/master`, or by creating a local branch out of it and checking this local branch out with `git checkout -b mylib_branch mylib_repo/master`. Alternatively, you can just list its files with `git ls-tree -r --abbrev mylib_repo/master`. You will see then that the subproject has a different project root from your superproject. Additionally, as seen from the `warning: no common commits` message, this remote-tracking branch contains a completely different history coming from a separate project.

Adding a subproject as a subtree

If you are not using specialized tools like `git subtree` but a manual approach, the next step will be a bit complicated and will require you to use some advanced Git concepts and techniques. Fortunately, it needs to be done only once.

First, if you want to import the subproject history, you would need to create a merge commit that will import the subproject in question. You

need to have the files of the subproject in the given directory in a superproject. Unfortunately, at least, with the current version of Git as of writing this chapter, using the `-Xsubtree=mylib/` merge strategy option would not work as expected. We would have to do it in two steps: prepare the parents and then prepare the contents.

The first step would then be to prepare a merge commit using the `ours` merge strategy, but without creating it (writing it to the repository). This strategy joins histories, but takes the current version of the files from the current branch:

```
$ git merge --no-commit --strategy=ours mylib_repo/master  
Automatic merge went well; stopped before committing as  
requested
```

If you want to have *simple history*, similar to the one we get from just copying files, you can skip this step.

We now need to update our index (the staging area for the commits) with the contents of the `master` branch from the library repository, and update our working directory with it. All this needs to happen in the proper subfolder too. This can be done with the low-level (plumbing) `git read-tree` command:

```
$ git read-tree --prefix=mylib/ -u mylib_repo/master  
$ git status  
On branch master  
All conflicts fixed but you are still merging.  
(use "git commit" to conclude merge)
```

Changes to be committed:

```
new file:   mylib/README  
[...]
```

We have used the `-u` option, so the working directory is updated along with the index.

Note

It is important to not forget the trailing slash in the argument of the `--`

prefix option. Checked out files are *literally* prefixed with it.

This set of steps is described in the HOWTO section of the Git documentation, namely in the *How to use the subtree merge strategy* moved earlier

<https://www.kernel.org/pub/software/scm/git/docs/howto/using-merge-subtree.html>.

It is much easier to use tools such as `git subtree`:

```
$ git subtree add --prefix=mylib mylib_repo master  
git fetch mylib_repo master  
Added dir 'mylib'
```

The `git subtree` command would fetch the subtree's remote when necessary; there's no need for the manual fetch that you had to perform in the manual solution.

If you examine the history, for example, with `git log --oneline --graph --decorate`, you will see that this command merged the library's history with the history of the application (of the superproject). If you don't want this, tough luck. The `--squash` option that `git subtree` offers on its `add`, `pull`, and `merge` subcommands won't help here. One of the peculiarities of this tool is that this option doesn't create a **squash merge**, but simply merges the squashed subproject's history (as if it were squashed with an interactive rebase). See, *Fig 2* later in the chapter.

If you want a subtree without its history attached to the superproject history, consider using `git-stree`. It has the additional advantage that it remembers the subtree settings and that it would create a remote if necessary:

```
$ git stree add mylib_repo -P mylib \  
https://git.example.com/mylib.git master  
warning: no common commits  
[master 5e28a71] [STree] Added stree 'mylib_repo' in mylib  
5 files changed, 32 insertions(+)  
create mode 100644 mylib/README  
[...]
```

```
STree 'mylib_repo' configured, 1st injection committed.
```

The information about the subtree's prefix (subdirectory), the branch, and so on is stored in the local configuration in the `stree.<name>` group. This stays in contrast to the behavior of `git subtree`, where you need to provide the prefix argument on each command.

Cloning and updating superprojects with subtrees

All right! Now that we have our project with a library embedded as a subtree, what do we need to do to get it? Because the concept behind subtrees is to have just one repository: the container, you can simply clone this repository.

To get an up-to-date repository you just need a regular pull; this would bring both superproject (the container) and subproject (the library) up to date. This works regardless of the approach taken, the tool used, and the manner in which the subtree was added. It is a great advantage of the subtrees approach.

Getting updates from subprojects with a subtree merge

Let's see what happens if there are some new changes in the subproject since we imported it. It is easy to bring the version embedded in the superproject up to date:

```
$ git pull --strategy subtree mylib_repo master
From https://git.example.com/mylib.git
 * branch           master    -> FETCH_HEAD
Merge made by the 'subtree' strategy.
```

You could have fetched and then merged instead, which allows for greater control. Or, you could have rebased instead of merging, if you prefer; that works too.

Note

Don't forget to select the merge strategy with `-s subtree` while pulling a subproject. Merging could work even without it, because Git does rename detection and would usually be able to discover that the files

were moved from the root directory (in the submodule) to a subdirectory (in the superproject we are merging into). The problematic case is when there are conflicting files inside and outside of the submodule. Potential candidates are `Makefiles` and other standard filenames.

If there are some problems with Git detecting the correct directory to merge into, or if you need advanced features of an ordinary recursive merge strategy (which is the default), you can instead use `-Xsubtree=<path/to/subproject>`, the `subtree` option of the recursive merge strategy.

You may need to adjust other parts of the application code to work properly with the updated code of the library.

Note that, with this solution, you have a submodule history attached to your application history, as you can see in *Fig 1*:

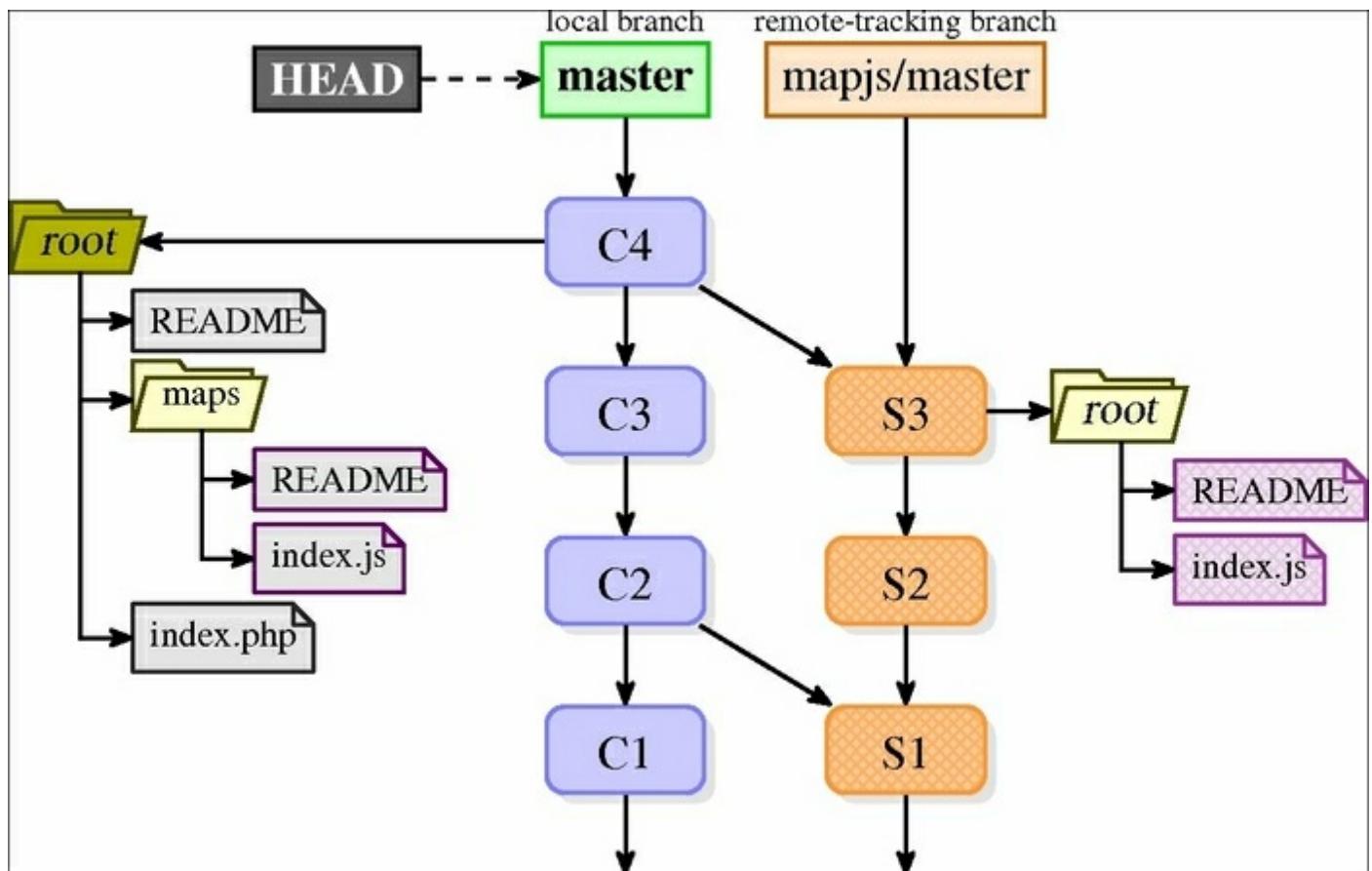


Fig 1: History of a superproject with a subtree-merged submodule

If you don't want to have the history of a subproject entangled in the history of a master project, and prefer a simple-looking history (as shown on *Fig. 2*), you can use the `--squash` option of `git merge` (or `git pull`) command to squash it.

```
$ git merge -s subtree --squash mylib_repo/master
Squash commit -- not updating HEAD
Automatic merge went well; stopped before committing as
requested
$ git commit -m "Updated the library"
```

In this case, you would have in the history only the fact that the version of the subproject had changed, which has its advantages and disadvantages. You get simpler history, but also simplified history.

With the `git subtree` or `git stree` tools, it is enough to use their `pull` subcommand; they supply the `subtree` merge strategy themselves. However, currently `git subtree pull` requires you to respecify `--prefix` and the entire subtree settings.

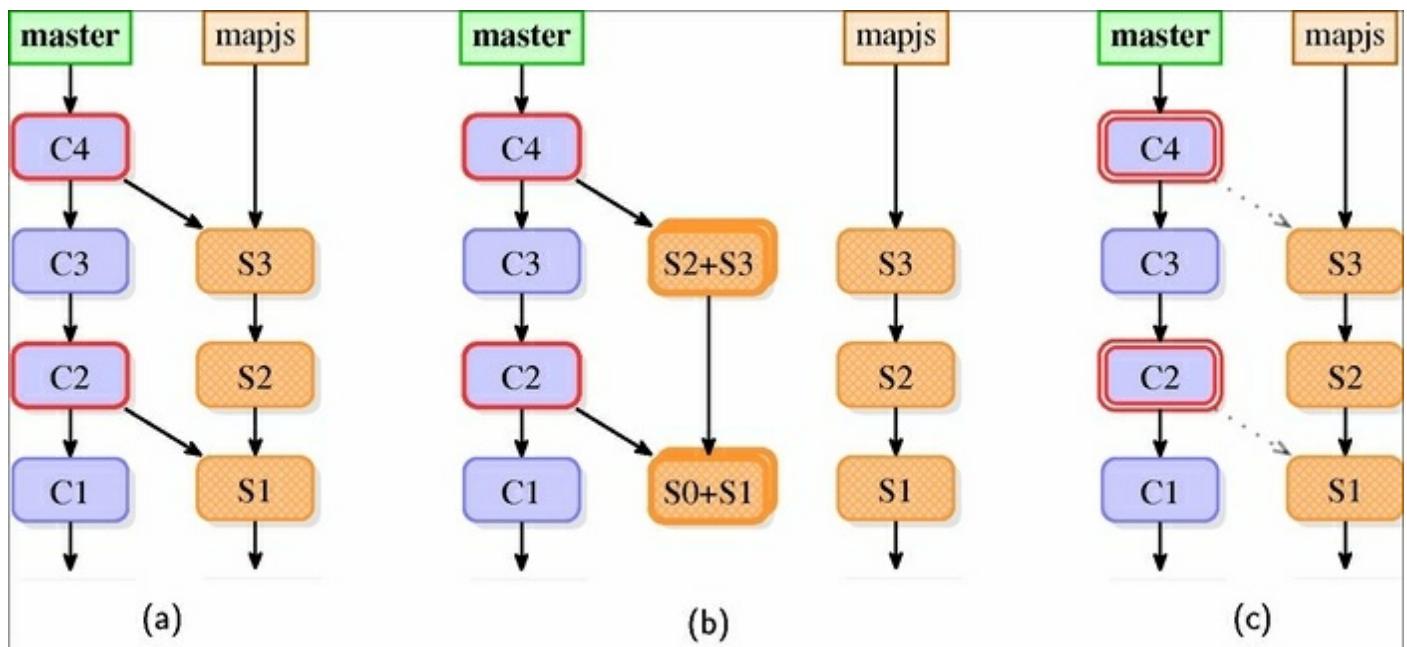


Fig 2: Different types of subtree merges: (a) subtree merge: `git pull -s subtree` and `git subtree pull`, (b) subtree merge of squashed commits: `git subtree pull --squash`, (c) squashed subtree merge: `git pull -s subtree --squash` and `git stree`. Note that dotted line in (c) denotes how commits `C2` and `C4` were made, and not that it is parent

commit.

Note that the `git subtree` command always merges, even with the `--squash` option; it simply squashes the subproject commits before merging (such as the `squash` instruction in the interactive rebase). In turn, `git stree pull` always squashes the merge (such as `git merge --squash`), which keeps the superproject history and subproject history separated without polluting the graph of the history. All this can be seen in *Fig 2*.

Showing changes between a subtree and its upstream

To find out the differences between the subproject and the current version in the working director, you need nontypical selector syntax for `git diff`. This is because all the files in the subproject (for example, in the `mylib_repo/master` remote-tracking branch) are in the root directory, while they are in the `mylib/` directory in the superproject (for example, in `master`). We need to select the subdirectory to be compared with `master`, putting it after the revision identifier and the colon (skipping it would mean that it would be compared with the root directory of the superproject).

The command looks as follows:

```
$ git diff master:mylib mylib_repo/master
```

Similarly, to check after the subtree merge whether the commit we just created (`HEAD`) has the same contents in the `mylib/` directory as the merged in commit, that is, `HEAD^2`, we can use:

```
$ git diff HEAD:mylib HEAD^2
```

Sending changes to the upstream of a subtree

In some cases, the subtree code of a subproject can only be used or tested inside the container code; most themes and plugins have such constraints. In this situation, you'll be forced to evolve your subtree code

straight inside the master project code base, before you finally backport it to the subproject upstream.

These changes often require adjustments in the rest of the superproject code; though it is recommended to make two separate commits (one for the subtree code change and one for the rest), it is not strictly necessary. You can tell Git to extract only the subproject changes. The problem is with the commit messages of the split changes, as Git is not able to automatically extract relevant parts of the changeset description.

Another common occurrence, which is best avoided but is sometimes necessary, is the need to customize the subproject's code in a container-specific way (configure it specifically for a master project), usually without pushing these changes back upstream. You should carefully distinguish between both the situations, keeping each use case's changes (backportable and nonbackportable) in their own commits.

There are different ways to deal with this issue. You can avoid the problem of extracting changes to be sent upstream by requiring that all the subtree changes have to be done in a separate module-only repository. If it is possible, we can even require that all the subproject changes have to be sent upstream first, and we can get the changes into the container only through upstream acceptance.

If you need to be able to extract the subtree changes, then one possible solution is to utilize `git filter-branch --directory-filter` (or `--index-filter` with the appropriate script). Another simple solution is to just use `git subtree push`. Both the methods, however, backport every commit that touches the subtree in question.

If you want to send upstream only a selection of the changes to the subproject of those that made it into the master project repository, then the solution is a bit more complicated. One possibility is to create a local branch meant specifically for backporting out of the subproject remote-tracking branch. Forking it from said subtree-tracking branch means that it has the subtree as the root and it would include only the submodule files.

This branch intended for backporting changes to the subproject would need to have the appropriate branch in the remote of the subproject upstream repository as its upstream branch. With such setup, we would then be able to `git cherry-pick --strategy=subtree` the commits we're interested in sending to the subproject's upstream onto it. Then, we can simply `git push` this branch into the subproject's repository.

Note

It is prudent to specify `--strategy=subtree` even if `cherry-pick` would work without it, to make sure that the files outside the subproject's directory (outside subtree) will get quietly ignored. This can be used to extract the subtree changes from the mixed commit; without this option, Git will refuse to complete the `cherry-pick`.

This requires much more steps than ordinary `git push`. Fortunately, you need to face this problem only while sending the changes made in the superproject repository back to the subproject. As you have seen, fetching changes from the subproject into the superproject is much, much simpler.

Well, it using `git-stree` would make this trivial: you just need to list the commits to be pushed to backport:

```
$ git stree push mylib_repo master~3 master~1
  5e28a71 [To backport] Support for creating debug symbols
  5b0aa4b [To backport] Timestamping (requires application
tweaks)
  STree 'mylib_repo' successfully backported local changes to
its remote
```

In fact, this tool uses internally the same technique, creating and using a backport-specific local branch for the subproject.

The Git submodules solution: repository inside repository

The subtrees method of importing the code (and possibly also history) of a subproject into the superproject has its disadvantages. In many cases,

the submodule and the container are two different projects: your application depends on the library, but it is obvious that they are separate entities. Joining the histories of the two doesn't look like the best solution.

Additionally, the embedded code and imported history of a submodule is always here. Therefore, the subtrees technique is not a good fit for optional dependencies and components (such as plugins or themes). It also doesn't allow you to have different access control for the submodule's history, with the possible exception of restricting write access to the submodule (actually to the subdirectory of a submodule), by using Git repository management solutions such as `gitolite` (you can find more in [Chapter 11, Git Administration](#)).

The submodule solution is to keep the submodule code and history in its own repository and to embed this repository inside the working area of a superproject, but not to add its files as superproject files.

Gitlinks, .git files, and the git submodule command

Git includes the command named `git submodule`, which is intended to work with submodules. Unfortunately, using this tool is not easy. To utilize it correctly, you need to understand at least some of the details of its operation. It is a combination of two distinct features: the so-called **gitlinks** and the `git submodule` tool itself.

Both in the subtree solution and the submodule solution, subprojects need to be contained in their own folder inside the working directory of the superproject. But while with subtrees the code of the submodule belongs to superproject repository, it is not the case for submodules. With submodules, each submodule has instead its own repository somewhere inside the working directory of its container repository. The code of the submodule belongs to its repository, and the superproject itself simply stores meta-information required to get appropriate revision of the submodule files.

In practice, in modern Git, submodules use a simple `.git` file with a

single `gitdir:` line containing a relative path to the actual repository folder. The submodule repository is actually located inside superproject's `.git/modules` folder (and has `core.worktree` set up appropriately). This is done mostly to handle the case when the superproject has branches that don't have submodule at all. It allows to avoid having to scrap the submodule's repository while switching to the superproject revision without it.

Note

You can think of the `.git` file with `gitdir:` line as a symbolic reference equivalent for the `.git` directories, an OS-independent symbolic link replacement. The path to the repository doesn't need to be a relative path.

```
$ ls -alOF plugins/demo/
total 10
drwxr-xr-x 1 user 0 Jul 13 01:26 ./
drwxr-xr-x 1 user 0 Jul 13 01:26 ../
-rw-r--r-- 1 user 32 Jul 13 01:26 .git
-rw-r--r-- 1 user 9 Jul 13 01:26 README
[...]
$ cat plugins/demo/.git
gitdir: ../../.git/modules/plugins/demo
```

Be that as it may, the contained superproject and the subproject module truly act as and, in fact, are independent repositories: they have their own history, their own staging area, and their own current branch. You should, therefore, take care while typing commands, minding if you're inside the submodule or outside it, because the context and impact of your commands differ drastically!

The main idea behind submodules is that the superproject commit remembers the exact revision of the subproject; this reference uses the SHA1 identifier of subproject commit. Instead of using a manifest-like file like in some dependency management tools, submodules solution stores this information in a tree object using the so-called gitlinks. Gitlink is a reference from a tree object (in the superproject repository) to a commit object (usually, in the submodule repository); see *Fig 3*.

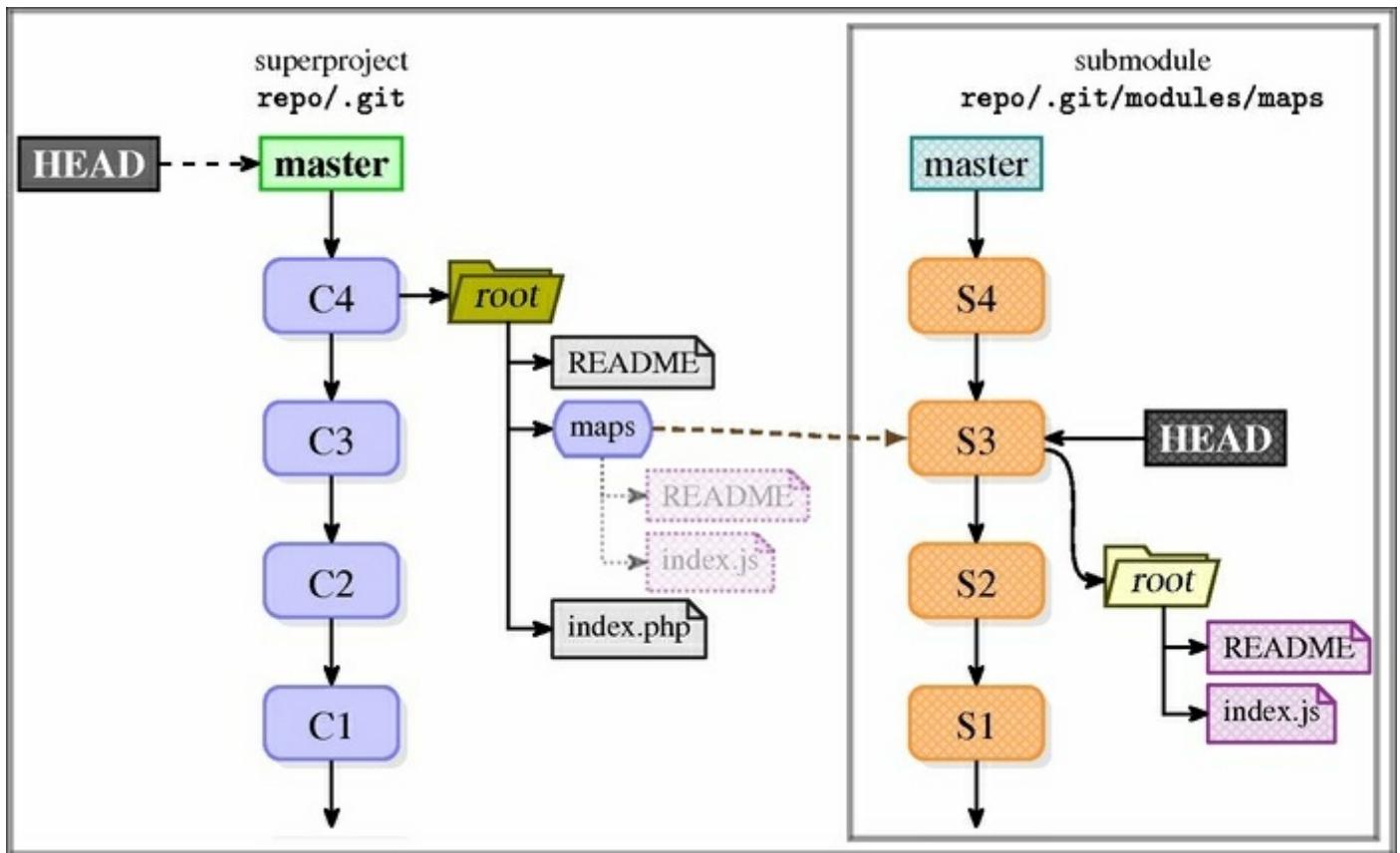


Fig 3: The history of a superproject with a subproject linked as a submodule . The faint shade of submodule files on left hand side denotes that there are present as files in the working directory of the superproject, but are not in the superproject repository themselves.

Recall that, following the description of the types of objects in the repository database from [Chapter 8, Keeping History Clean](#), each commit object (representing a revision of a project) points exactly to one tree object with the snapshot of the repository contents. Each tree object references blobs and trees, representing file contents and directory contents, respectively. The tree object referenced by the commit object uniquely identifies the set of files contents, file names, and file permissions contained in a revision associated with the commit object.

Let's remember that the commit objects themselves are connected with each other, creating the Directed Acyclic Graph (DAG) of revisions. Each commit object references zero or more parent commits, which together describe the history of a project.

Each type of the references mentioned earlier took part in the reachability check. If the object pointed to was missing, it means that the repository is corrupt.

It is not so for gitlinks. Entries in the tree object pointing to the commits refer to the objects in the other separate repository, namely in the subproject (submodule) repository. The fact that the submodule commit being unreachable is not an error is what allows us to optionally include submodules; no submodule repository, no commit referenced in gitlink.

The results of running `git ls-tree --abbrev HEAD` on a project with all the types of objects is as follows:

```
040000 tree 573f464      docs
100755 blob f27adc2     executable.sh
100644 blob 1083735     README.txt
040000 tree ef9bcb4     subdirectory
160000 commit 5b0aa4b    submodule
120000 blob 3295d66     symlink
```

Compare it with the contents of the working area (with `ls -l -o -F`):

```
drwxr-xr-x  5 user       12288 06-28 17:18 docs/
-rwxr-xr-x  1 user       36983 02-20 20:11 executable.sh*
-rw-r--r--  1 user        2628 2015-01-03 README.txt
drwxr-xr-x  3 user        4096 06-28 17:19 subdirectory/
drwxr-xr-x 48 user       36864 06-28 17:19 submodule/
lrwxrwxrwx  1 user        32 06-28 17:18 symlink ->
docs/toc.html
```

Adding a subproject as a submodule

With subtrees, the first step was usually to add a subproject repository as a remote, which meant that objects from the subproject repository were fetched into the superproject object database.

With submodules, the subproject repository is kept separate. You could manage cloning the subproject repository manually from inside the superproject worktree and then add the gitlink also by hand with `git add <submodule directory>` (without a trailing slash).

Note

Important note!

Normally, commands `git add subdir` and `git add subdir/` (the latter with a forward slash, which following the POSIX standard denotes a subdirectory) are equivalent. This is not true if you want to create `gitlink`! If `subdir` is a top directory of an embedded Git repository of a subproject, the former would create a `gitlink` reference, while the latter in the form of `git add subdir/` would add all the files in the `subdir` individually, which is not probably what you expect.

A simpler and better solution is to use the `git submodule` command, which was created to help manage the filesystem contents, the metadata, and the configuration of your submodules, as well as inspect their status and update them. To add the given repository as a submodule at a specific directory in the superproject, use the `add` subcommand of the `git submodule`:

```
$ git submodule add https://git.example.com/demo-plugin.git \
  plugins/demo
Cloning into 'plugins/demo'...
done.
```

Note

Note:

While using paths instead of URLs for remotes, you need to remember that the relative paths for remotes are interpreted relative to our main remote, not to the root directory of our repository.

This command stores the information about the submodule, for example the URL of the repository, in the `.gitmodules` file. It creates this file if it does not exist:

```
[submodule "plugins/demo"]
  path = plugins/demo
  url = https://git.example.com/demo-plugin.git
```

Note that a submodule gets a name equal to its path. You can set the name explicitly with the `--name` option (or by editing the configuration); `git mv` on a submodule directory will change the submodule path but keep the same name.

Tip

Reuse of authentication while fetching submodules

While storing the URL of a remote repository, it is often acceptable and useful to store the username with the subproject information (for example, storing the username in a URL, like `user@git.company.com:mylib.git`).

However, remembering the username as a part of URL is undesirable in `.gitmodules`, because this file must be visible by other developers (which often use different usernames for authentication). Fortunately, the commands that descend into submodules can reuse the authentication from cloning (or fetching) a superproject.

The `add` subcommand also runs an equivalent of `git submodule init` for you, assuming that if you have added a submodule, you are interested in it. This adds some submodule-specific settings to the local configuration of the master project:

```
[submodule "plugins/demo"]
  url = https://git.example.com/demo-plugin.git
```

Why the duplication? Why store the same information in `.gitmodules` and in `.git/config`? Well, because while the `.gitmodules` file is meant for all developers, we can fit our local configuration to specific local circumstances. The other reason for using two different files is that while the presence of the submodule information in `.gitmodules` means only that the subproject is available, having it also in `.git/config` implies that we are interested in a given submodule (and that we want it to be present).

You can create and edit the `.gitmodules` file by hand or with `git`

`config -f .gitmodules`. This is useful if, for example, you have added a submodule by hand by cloning it, but want to use `git submodule` from now on.

This file is usually committed to the superproject repository (similar to `.gitignore` and `.gitattributes` files), where it serves as the list of possible subprojects.

Note

All the other subcommands require this file to be present; for example, if we would run `git submodule update` before adding it, we would get:

```
$ git submodule update  
No submodule mapping found in .gitmodules for path  
'plugins/demo'
```

That's why `git submodule add` stages both the `.gitmodules` file and the submodule itself:

```
$ git status  
On branch master  
Changes to be committed:  
(use "git reset HEAD <file>..." to unstage)  
  
  new file:   .gitmodules  
  new file:   plugins/demo
```

Note that the whole submodule, which is a directory, looks to the git status like the new file. By default, most Git commands are limited to the active container repository only, and do not descent to the nested repositories of the submodules. As we will see, this is configurable.

Cloning superprojects with submodules

One important issue is that, by default, if you clone the superproject repository, you would not get any submodules. All the submodules will be missing from the working duplicated directory; only their base directories are here. This behavior is the basis of the optionality of submodules.

We need then to tell Git that we are interested in a given submodule. This is done by calling the `git submodule init` command. What this command does is it copies the submodule settings from the `.gitmodules` file into the superproject's repository configuration, namely, `.git/config`, registering the submodule:

```
$ git submodule init plugins/demo
Submodule 'plugins/demo' (https://git.example.com/demo-
plugin.git) registered for path 'plugins/demo'
```

The `init` subcommand adds the following two lines to the `.git/config` file:

```
[submodule "plugins/demo"]
  url = https://git.example.com/demo-plugin.git
```

This separate local configuration for the submodules you are interested in allows you also to configure your local submodules to point to a different location URL (perhaps, a per-company reference clone of a subproject's repository) than the one that is present in `.gitmodules` file.

This mechanism also makes it possible to provide the new URL if the repository of a subproject moved. That's why the local configuration overrides the one that is recorded in `.gitmodules`; otherwise you would not be able to fetch from current URL when switched to the version before the URL change. On the other hand, if the repository moved, and the `.gitmodules` file was updated accordingly, we can re-extract new URL from `.gitmodules` into local configuration with `git submodule sync`.

We have told Git that we are interested in the given submodule. However, we have still not fetched the submodule commits from its remote and neither have we checked it out and have its files present in the working directory of the superproject. We can do this with `git submodule update`.

Note

In practice, while dealing with submodule using repositories, we usually

group the two commands (`init` and `update`) into one with `git submodule update --init`.

well, at least if we don't need to customize the URL.

If you are interested in all the submodules, you can use `git clone --recursive` to automatically initialize and update each submodule right after cloning.

To temporarily remove a submodule, retaining the possibility of restoring it later, you can mark it as not interesting with `git remote deinit`. This just affects `.git/config`. To permanently remove a submodule, you need to first deinit it and then remove it from `.gitmodules` and from the working area (with `git rm`).

Updating submodules after superproject changes

To update the submodule so that the working directory contents reflect the state of a submodule in the current version of superproject, you need to perform `git submodule update`. This command updates the files of the subproject or, if necessary, clones the initial submodule repository:

```
$ rm -rf plugins/demo    # clean start for this example  
$ git submodule update  
Submodule path 'plugins/demo': checked out '5e28a713d8e87...'
```

The `git submodule update` command goes to the repository referenced by `.git/config`, fetches the ID of the commit found in the index (`git ls-tree HEAD -- plugins/demo`), and checks out this version into the directory given by `.git/config`. You can, of course, specify the submodule you want to update, giving the path to the submodule as a parameter.

Because we are here checking out the revision given by gitlink, and not by a branch, `git submodule update` detaches the subprojects' `HEAD` (see Fig 3). This command rewinds the subproject straight to the version recorded in the supermodule.

There are a few more things that you need to know:

- If you are changing the current revision of a superproject in any way, either by changing a branch, by importing a branch with `git pull`, or by rewinding the history with `git reset`, you need to run `git submodule update` to get the matching content to submodules. This is not done automatically, because it could lead to potentially losing your work in a submodule.
- Conversely, if you switch to another branch, or otherwise change the current revision in a superproject, and do not run `git submodule update`, Git would consider that you changed your submodule directory deliberately to point to a new commit (while it is really an old commit, that you used before, but you forgot to update). If, in this situation, you would run `git commit -a`, then by accident, you will change gitlink, leading to having an incorrect version of a submodule stored in the superproject history.
- You can upgrade the gitlink reference simply by fetching (or switching to) the version of a submodule you want to have by using ordinary Git commands inside the subproject, and then committing this version in the supermodule. You don't need to use the `git submodule` command here.

You can have Git to automatically fetch the initialized submodules while pulling the updates from the master project's remote repository. This behavior can be configured using `fetch.recurseSubmodules` (or `submodule.<name>.fetchRecurseSubmodules`). The default value for this configuration is `on-demand` (to fetch if gitlink changes, and the submodule commit it points to is missing). You can set it to `yes` or `no` to turn recursively fetching submodules on or off unconditionally. The corresponding command-line option is `--recurse-submodules`.

It is however critical to remember that even though Git can automatically fetch submodules, it *does not auto-update*. Your local clone of the submodule repository is up to date with the submodule's remote, but the submodule's working directory is stuck to its former contents. If you don't explicitly update the submodule's working directory, the next commit in the container repository will regress the submodule. Currently, there are no configuration settings or command-

line options that can autoupdate all the auto-fetched submodules on pull. Well, there were no such options at the time of this writing, but hopefully the management of submodules in Git will improve.

Note that instead of checking out the gitlinked revision on detached HEAD, we can merge the commit recorded in the superproject into the current branch in the submodule with `--merge`, or rebase the current branch on top of the gitlink with `--rebase`, just like with `git pull`. The submodule repository branch used defaults to master, but the branch name may be overridden by setting the `submodule.<name>.branch` option in either `.gitmodules` or `.git/config`, the latter taking precedence.

As you can see, using gitlinks and the `git submodule` command is quite complicated. Fundamentally, the concept of gitlink might fit well to the relationship between subprojects and your superproject, but using this information correctly is harder than you think. On the other hand, it gives great flexibility and power.

Examining changes in a submodule

By default, the status, logs, and diff output is based solely on the state of the active repository, and does not descend into submodules. This is often problematic; you would need to remember to run `git submodule summary`. It is easy to miss a regression if you are limited to this view: you can see that the submodule has changed, but you can't see how.

You can, however, set up Git to make it use a *submodule-aware status* with the `status.submoduleSummary` configuration variable. If it is set to a nonzero number, this number will provide the `--summary-limit` restriction; a value of `true` or `-1` will mean an unlimited number.

After setting this configuration, you would get something like the following redundant:

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
```

```
new file:   .gitmodules
new file:   plugins/demo
```

Submodule changes to be committed:

```
* plugins/demo 0000000...5e28a71 (3):
  > Fix repository name in a README file
```

The status extends the always present information that the submodule changed (new file: plugins/demo), adding the information that the submodule present at plugins/demo got three new commits, and showing the summary for the last one (Fix repository name in a README file). The right pointing angle bracket > preceding the summary line means that the commit was added, that is, present in the working area but not (yet) in the superproject commit.

Note

Actually, this added part is just the git submodule summary output.

For the submodule in question, a series of commits in the submodule between the submodule version in the given superproject's commit and the submodule version in the index or the working tree (the former shown by using --cached) are listed. There is also git submodule status for short information about each module.

The git diff command's default output also doesn't tell much about the change in the submodule, just that it is different:

```
$ git diff HEAD -- plugins/demo
diff --git a/plugins/demo b/plugins/demo
new file mode 160000
index 0000000..5e28a71
--- /dev/null
+++ b/plugins/demo
@@ -0,0 +1 @@
+Subproject commit 5e28a713d8e875f2cf1060c2580886dec3e5b04c
```

Fortunately, there is the --submodule=log command-line option (that you can enable by default with the diff.submodule configuration

setting) that lets us see something more useful:

```
$ git diff HEAD --submodule=log -- plugins/demo
Submodule subrepo 000000...5e28a71 (new submodule)
```

Instead of using `log`, we can use the `short` format that shows just the names of the commits, which is the default if the format is not given (that is, with just `git diff --submodule`).

Getting updates from the upstream of the submodule

To remind you, the submodule commits are referenced in gitlinks using the SHA1 identifier, which always resolves to the same revision; it is not a volatile (inconstant) reference such as a branch name. Because of this, a submodule in a superproject does not automatically upgrade (which could possibly be breaking the application). But sometimes you may want to update it.

Let's assume that the subproject repository got new revisions published and we want, for our superproject, to update to the new version of a submodule.

To achieve this, we need to update the local repository of a submodule, move the version we want to the working directory of the superproject, and finally commit the submodule change in the superproject.

We can do this manually, starting by first changing current directory to be inside the working directory of the submodule. Then, inside the submodule, we perform `git fetch` to get the data to the local clone of the repository (in `.git/modules/` in the superproject). After verifying what we have with `git log`, we can then update the working directory. If there are no local changes, you can simply checkout the desired revision. Finally, you need to create a commit in a superproject.

In addition to the finer-grained control, this approach has the added benefit of working regardless of your current state (whether you are on an active branch or on a detached HEAD).

Another way to go about this would be, working from the container repository, to explicitly upgrade the submodule to its tracked remote branch with `git submodule update --remote`. Similarly to the ordinary `update` command, you can choose to merge or rebase instead of checking out a branch; you can configure the default way of updating with the `submodule.<name>.update` configuration variable, and the default upstream branch with `submodule.<name>.branch`.

Note

In short, `submodule update --remote --merge` will merge upstream's subproject changes into the submodule, while `submodule update --merge` will merge the superproject gitlink changes into the submodule.

The `git submodule update --remote` command would fetch new changes from the submodule remote site automatically, unless told not to with `--no-fetch`.

Sending submodule changes upstream

One of the major dangers in making changes live directly in a submodule (and not via its standalone repository) is forgetting to push the submodule. A good practice for submodules is to commit changes to the submodule first, push the module changes, and only then get back to the container project, commit it, and push the container changes.

If you only push to the supermodule repository, forgetting about the submodule push, then other developers would get an error while trying to get the updates. Though Git does not complain while fetching the superproject, you would see the problem in the `git submodule summary` output (and in the `git status` output, if properly configured) and while trying to update the working area:

```
$ git submodule summary
* plugins/demo 12e3a52...0e90143:
  Warn: plugins/demo doesn't contain commit
  12e3a529698c519b2fab790...
$ git submodule update
fatal: reference is not a tree: 12e3a529698c519b2fab790...
```

Unable to checkout '12e3a529698c519b2fab790...' in submodule path
'plugins/demo'

You can plainly see how important it is to remember to push the submodule. You can ask Git to automatically push the submodules while pushing the superproject, if it is necessary, with `git push --recurse-submodules=on-demand` (the other option is just to check). With Git 2.7.0 or later you can also use the `push.recurseSubmodules` configuration option.

Transforming a subfolder into a subtree or submodule

The first issue that comes to mind while thinking of the use cases of subprojects in Git is about having source code of the base project be ready for such division. Submodules and subtrees are always expressed as subdirectories of the superproject (the master project). You can't mix files from different subsystems in one directory.

Experience shows that most systems use such a directory hierarchy, even in monolithic repositories, which is a good beginning for modularization efforts. Therefore, transforming a subfolder into a real submodule/subtree is fairly easy and can be done in the following sequence of steps:

1. Move the subdirectory in question outside the working area of a superproject to have it beside the top directory of superproject. If it is important to keep the history of a subproject, consider using `git filter-branch --subdirectory-filter` or its equivalent, perhaps together with tools such as `reposurgeon` to clean up the history. See [Chapter 8, Keeping History Clean](#) for more details.
2. Rename the directory with the subproject repository to better express the essence of the extracted component. For example, a subdirectory originally named `refresh` could be renamed to `refresh-client-app-plugin`.
3. Create the public repository (upstream) for the subproject, as a first class project (for example, create a new project on GitHub to keep

extracted code, either under the same organization as a superproject, or under a specialized organization for application plugins).

4. Initialize now a self-sufficient and standalone plugin as a Git repository with `git init`. If in step 1 you have extracted the history of the subdirectory into some branch, then push this branch into the just created repository. Set up the public repository created in step 3 as a default remote repository and push the initial commit (or the whole history) to the just created URL to store the subproject code.
5. In the superproject, read the subproject you have just extracted; this time, as a proper submodule or subtree, whichever solution is a better fit and whichever method you prefer to use. Use the URL of the just created public repository for the subproject.
6. Commit the changes in the superproject and push them to its public repository, in the case of submodules including the newly created (or the just modified) `.gitmodules` file.

The recommended practice for the transformation of a subdirectory into a standalone submodule is to use a read-only URL for cloning (adding back) a submodule. This means that you can use either the `git://` protocol (warning: in this case the server is unauthenticated) or `https://` without a username. The goal of this recommendation is to enforce separation by moving the work on a submodule code to a standalone separate subproject repository. In order to ensure that the submodule commits are available to all other developers, every change should go through the public repository for a subproject.

If this recommendation (best practice) is met with a categorical refusal, in practice you could work on the subproject source code directly inside the superproject, though it is more error prone. You would need to remember to commit and push in the submodule first, doing it from inside of the nested submodule subdirectory; otherwise other developers would be not able to get the changes. This combined approach might be simpler to use, but it loses the true separation between implementing and consuming changes, which should be better assumed while using submodules.

Subtrees versus submodules

In general, subtrees are easier to use and less tricky. Many people go with submodules, because of the better built-in tooling (they have their own Git command, namely `git submodule`), detailed documentation, and similarity to the Subversion externals, making them feel falsely familiar. Adding a submodule is very simple (just run `git submodule add`), especially compared to adding a subtree without the help of third-party tools such as `git subtree` or `git stree`.

The major difference between subtrees and submodules is that, with subtrees, there's only one repository, which means just one lifecycle. Submodules and similar solutions use nested repositories, each with its own lifeline.

Though submodules are easy to set up and fairly flexible, they are also fraught with peril, and you need to practice vigilance while working with them. The fact that the submodules are opt-in also means that the changes touching the submodules demand a manual update by every collaborator. Subtrees are always there, so getting the superproject's changes mean getting the subproject's too.

Commands such as `status`, `diff`, and `log` display precious little information about submodules, unless properly configured to cross the repository boundary; it is easy to miss a change. With subtrees, `status` works normally, while `diff` and `log` need some care, because the subproject commits have a different root directory. The latter assumes that you did not decide to not include the subproject history (by squashing subtree merges). Then, the problem is only with the remote-tracking branches in subproject's repository, if any.

Because the lifecycles of different repositories are separate, updating a submodule inside its containing project requires two commits and two pushes. Updating a subtree-merged subproject is very simple: only one commit and one push. On the other hand, publishing the subproject changes upstream is much easier with submodules, while it requires

changeset extraction with subtrees (here tools such as `git subtree` help a lot).

The next major issue, and a source of problems, is that the submodule has two sources of the current revision: the gitlink in the superproject and the branches in the submodule's clone of the repository. This means that `git remote update` works a bit like a sideways push into a nonbare repository (see [Chapter 6, Advanced Branching Techniques](#)).

Submodule heads are therefore generally detached, so any local update requires various preparatory actions to avoid creating a lost commit.

There is no such issue with subtrees. All the revision changing commands work as usual with subtrees, bringing the subproject directory to the correct version without the requirement of any additional action. Getting changes from the subproject repository is just a subtree merge away. The only difference between ordinary pull is the `-s subtree` option.

Still, sometimes submodules are the right choice. Compared to subtrees, they allow for a subproject (a module) to be not fetched, which is helpful when your code base is massive. Submodules are also useful when the heavy modularization is not natively handled, or not well natively handled, by the development stack's ecosystem.

Submodules might also themselves be superprojects for other submodules, creating a hierarchy of subprojects. Using nested submodules is made easier thanks to `git submodule status`, `update`, `foreach`, and `sync` subcommands all supporting the `--recursive` switch.

Use cases for subtrees

With subtrees, there is only one repository, no nested repositories, just like a regular codebase. This means that there is just one lifecycle. One of the key benefits of subtrees is being able to mix container-specific customizations with general purpose fixes and enhancements.

Projects can be organized and grouped together in whatever way you find to be most logically consistent. Using a single repository also

reduces the overhead from managing dependencies.

The basic example of using subtrees is managing the customized version of a library, a required dependency. It is easy to get a development environment set up to run builds and tests. Monorepo makes it also viable to have one universal version number for all the projects. Atomic cross-submodule commits are possible; therefore, a repository can always be in a consistent state.

You can also use subtrees for embedding related projects, such as a GUI or a web interface, inside a superproject. In fact, many use cases for submodules can also apply to the subtrees solution, with an exception of the cases where there is a need for a subproject to be optional, or to have different access permissions than a master project. In those cases you need to use submodules.

Use cases for submodules

The strongest argument for the use of submodules is the issue of modularization. Here, the main area of use for submodules is handling plugins and extensions. Some programming ecosystems, such as ANSI C and C++, and also Objective-C, lack good and standard support for managing version-locked multimodule projects. In this case, a plugin-like code can be included in the application (superproject) using submodules, without sacrificing the ability to easily update to the latest version of a plugin from its repository. The traditional solution of putting instructions about how to copy plugins in the `README`, disconnects it from the historical metadata.

This schema can be extended also to the noncompiled code, such as the Emacs List settings, configuration in dotfiles, (including frameworks such as `oh-my-zsh`), and themes (also for web applications). In these situations, what is usually needed to use a component is the physical presence of a module code at conventional locations inside the master project tree, which are mandated by the technology or framework being used. For instance, themes and plugins for Wordpress, Magento, and so on are often de facto installed this way. In many cases, you need to be in

a superproject to test these optional components.

Yet another particular use case for submodules is the division based on access control and visibility restriction of a complex application. For example, the project might use a cryptographic code with license restrictions, limiting access to it to the small subset of developers. With this code in a submodule with restricted access to its repository, other developers would simply be unable to clone this submodule. In this solution, the common build system needs to be able to skip cryptographic component if it is not available. On the other hand, the dedicated build server can be configured in such a way that the client gets the application build with crypto enabled.

A similar visibility restriction purpose, but in reverse, is making the source code of examples available long before it was to be published. This allows for better code thanks to the social input. The main repository for a book itself can be closed (private), but having an `examples/` directory contain a submodule intended for a sample source code allows you to make this subrepository public. While generating the book in the PDF and EPUB (and perhaps also MOBI) formats, the build process can then embed these examples (or fragments of them), as if they were ordinary subdirectory.

Third-party subproject management solutions

If you don't find a good fit in either `git subtree` or `git submodule`, you can try to use one of the many third-party projects to manage dependencies, subprojects, or collections of repositories. One such tool is the `externals` (or `ext`) project by Miles Georgie. You can find it at <http://nopugs.com/ext-tutorial>. This project is VCS-agnostic, and can be used to manage any combination of version control systems used by subprojects and superprojects.

Another is the `repo` tool (<https://android.googlesource.com/tools/repo/>) used by the Android Open Source project to unify the many Git repositories for across-network operations. You can find many other such tools.

Note

When choosing between native support and one of the many tools to manage many repositories together, you should check whether the tool in question uses a subtree-like or submodule-like approach to find if it would be a good fit for your project.

Managing large Git repositories

Because of its distributed nature, Git includes the full change history in each copy of the repository. Every clone gets not only all the files, but every revision of every file ever committed. This allows for efficient development (local operations not involving a network are usually fast enough so that they are not a bottleneck) and efficient collaboration with others (distributed nature allows for many collaborative workflows).

But what happens when the repository you want to work on is really huge? Can we avoid taking a large amount of disk space for version control storage? Is it possible to reduce the amount of data that end users need to retrieve while cloning the repository?

If you think about it, there are broadly two main reasons for repositories to grow massive: they can accumulate a very long history (the every revision direction), or they can include huge binary assets that need to be managed together with code (the every file direction), or both. For those two scenarios, the techniques and workarounds are different, and can be applied independently.

Handling repositories with a very long history

Even though Git can effectively handle repositories with a long history, very old projects spanning huge number of revisions can become a pain to clone. In many cases, you are not interested in ancient history and do not want to pay the time to get all the revisions of a project and the disk space to store them.

For example, if you want to propose a new feature or a bugfix (the latter might require running `git bisect` on your machine, where the regression bug is easily reproducible; see [Chapter 2, Exploring Project History](#) for how to use bisection), you don't want to wait for the full clone to finish, which may take quite a while.

Note

Some Git repository hosting services, such as GitHub, offer a web-based interface to manage repositories, including in-browser file management and editing. They may even automatically create a fork of the repository for you to enable writing and proposing changes.

But a web-based interface doesn't cover everything, and you might be using self-hosted repositories or a service that doesn't provide this feature anyway.

Using shallow clones to get truncated history

The simple solution to a fast clone and to saving disk space is to perform shallow clone using Git. This operation allows you to get a local copy of the repository with the history truncated to a particular specified depth, that is, the number of latest revisions.

How do you do it? Just use the `--depth` option:

```
$ git clone --depth=1 https://git.company.com/project
```

The preceding command clones only the most recent revision of the primary branch. This trick can save quite a bit of time and relieve a great deal of load from the servers. Often, shallow clone finishes in seconds rather than in minutes; a significant improvement.

Since version 1.9, Git supports pull and push operations even with shallow clones, though some care is still required. You can change the depth of a shallow clone by providing the `--depth=<n>` option to `git fetch` (note however that tags for the deepened commits are not fetched). To turn a shallow repository into a complete one, use `--unshallow`.

Note, also `git clone --depth=1` may still get all the branches and all the tags. This can happen if the remote repository doesn't have `HEAD`, thus it doesn't have a primary branch selected; otherwise only the tip of the said single branch is fetched. Long-lived projects usually had many

releases during their long history. To really save time, you would need then to combine shallow clone with the next solution: branch limiting.

Cloning only a single branch

Git, by default, clones all the branches and tags (if you want to fetch notes or replacements, you need to specify them explicitly). You can limit the amount of the history you clone by specifying that you want to clone only a single branch:

```
$ git clone --branch master --single-branch \
https://git.company.com/project
```

Because most of the project history (most of the DAG of revisions) is shared among branches, with very few exceptions, you probably won't see a huge difference using this.

This feature might be quite useful if you don't want detached orphan branches or the opposite: you want only an orphan branch (for example, with a web page for a project). It also works well used together with a shallow clone.

Handling repositories with large binary files

In some specific circumstances, you might need to track *huge binary assets* in the code base. Gaming teams have to handle huge 3D models, web development teams might need to track raw image assets or Photoshop documents, and both might require having video files under version control. Sometimes, you might want the convenience of including large binary deliverables that are difficult or expensive to generate, for example, storing a snapshot of a virtual machine image.

There are some tweaks to improve the handling of binary assets by Git. For binary files that change significantly from version to version (and not just change some metadata headers), you might want to turn off the delta compression by adding `-delta` explicitly for specific types of files in a `.gitattributes` file (see [Chapter 4, Managing Your Worktree](#), and [Chapter 10, Customizing and Extending Git](#)). Git would automatically

turn off delta compression for any file above the `core.bigFileThreshold` size, 512 MiB by default. You might also want to turn the compression off (for example if a file is in the compressed format already); though because `core.compression` and `core.looseCompression` are global for the whole repository, it makes more sense if binary assets are in a separate repository (submodule).

Splitting the binary asset folder into a separate submodule

One possible way of handling large binary asset folders is, as mentioned earlier, to split them into a separate repository and pull the assets into your main project as submodule. The use of submodules gives a way to control when assets are updated. Moreover, if a developer does not need those binary assets to work, he or she can simply exclude the submodule with assets from fetching.

The limitation is that you need to have a separate folder with these huge binary assets that you want to handle this way.

Tip

Sparse checkout

Git includes the technique that allows you to explicitly detail which files and folders you want to populate on checkout. This mode is turned on by setting the `core.sparseCheckout` configuration variable to `true`, and uses the `.git/info/sparse-checkout` file with the `gitignore` syntax to specify what is to appear in the working directory. The index is populated in full, with `skip-worktree` set for files missing from the checkout.

While it can be helpful if you have a huge tree of folders, it doesn't affect the overall size of the local repository itself.

Storing large binary files outside the repository

Another solution is to use one of the many third-party tools that try to solve the problem of handling large binary files in Git repositories. Many

of them are using a similar paradigm, namely, storing the contents of huge binary files outside the repository, while providing some kind of pointers to the contents in the checkout.

There are three parts of each such implementation: how they store the information about the contents of the managed files inside the repository, how they manage sharing the large binary files between a team, and how they integrate with Git (and what is their performance penalty). While choosing a solution, you would need to take this data into account, together with the operating system support, ease of use, and the size of the community.

What is stored in the repository and what is checked in might be a *symlink* to the file or to the key, or it might be a *pointer file* (often plain text), which acts as a reference to the actual file contents (by name or by the cryptographic hash of file contents). The tracked files need to be stored in some kind of backend for a collaboration (cloud service, rsync, shared directory, and so on). Backends might be accessed directly by the client, or there might be a separate server with a defined API into which the blobs are written to, which would in turn offload the storage elsewhere.

The tool might either require the use of separate commands for checking out and committing large files and for fetching from and pushing to the backend, or it might be integrated into Git. The integrated solution uses the `clean/smudge` filters to handle check-out and check-in transparently, and the `pre-push` hook to send large file contents transparently together. You only need to state which files to track and, of course, initialize the repository for the tool use.

The advantage of a filter-based approach is the ease of use; however, there is a performance penalty, because of how it works. Using separate commands to handle large binary assets makes the learning curve a bit steeper, but provides for better performance. Some tools provide both interfaces.

Among different solutions, there are **git annex** with a large community

and support for various backends, and Git LFS (Large File Storage) created by GitHub, which provides good MS Windows support, client-server approach, and transparency (with support for filter-based approach). There are many other such tools, for example, **git-fat**, **git-media**, **git-bigstore**, and **git-sym**.

Summary

This chapter provided all the tools you need to manage multicomponent projects with Git, from libraries and graphical interfaces, through plugins and themes, to frameworks.

You learned the concept behind the subtrees technique and how to use it to manage subprojects. You know how to create, update, examine, and manage subprojects using subtrees.

You got to know the submodule approach of nested repositories for optional dependencies. You learned the ideas behind gitlinks, `.gitmodules`, and the `.git` files. You encountered the pitfalls and traps for the unwary that you need to be vigilant about while using submodules. You know the reason for these problems and understand the notions behind them. You know how to create, update, examine, and manage subprojects using submodules.

You learned when to use subtrees and submodules, and their advantages and disadvantages. You know a few use cases for each technique.

Now that you know how to use Git effectively in a variety of circumstances, and learned the high-level ideas behind Git behavior that helps you understand it, it's time to tackle how to make Git easier to use in [Chapter 10, *Customizing and Extending Git*](#).

Chapter 10. Customizing and Extending Git

Earlier chapters were designed to help you understand and master Git as a version control system, from examining history, through managing your contributions, to collaborating with other developers, ending with handling the composite projects in the last chapter: [Chapter 9, *Managing Subprojects—Building a Living Framework*](#).

The following two chapters would help set up and configure Git, so that you can use it more effectively for yourself (this chapter) and help other developers use it (the next chapter).

This chapter will cover configuring and extending Git to fit one's needs. First, it will show how to set up a Git command line to make it easier to use. For some tasks though it is easier to use visual tools; the short introduction to graphical interfaces in this chapter should help you in choosing one. Next, there will be an explanation on how to change and configure Git behavior, from configuration files (with the selected configuration options described), to a per-file configuration with the `gitattributes` file.

Then this chapter will cover how to automate Git with hooks, describing for example how to make Git check whether the commit being created passes coding guidelines for a project. This part will focus on the client-side hook, and will only touch upon the server-side hooks—those are left for the, [Chapter 11, *Git Administration*](#). The last part of the chapter will describe how to extend Git, from the Git command aliases, through integrating new user-visible commands, to helpers and drivers (new back-end abilities).

Many issues, such as `gitattributes`, remote and credential helpers, and the basics of the Git configuration should be known from the previous chapters. This chapter will gather this information in a single place and expand it a bit.

In this chapter, we will cover the following topics:

- Setting up shell prompt and TAB completion for a command line
- Types and examples of graphical user interfaces
- Configuration files and basic configuration options
- Installing and using various types of hooks
- Simple and complex aliases
- Extending Git with new commands and helpers

Git on the command line

There are a lot of different ways to use the Git version control system. There are many graphical user interfaces (GUIs) of varying use cases and capabilities, and there exists tools and plugins that allow integration with an integrated development environment (IDE) or a file manager.

However, the command line is the only place you can run all of the Git commands and which provides support for all their options. New features, which you might want to use, are developed for the command line first. Also, most of the GUIs implement only some subsets of the Git functionality. Mastering the command line always guarantees a deep understanding of tools, mechanisms, and their abilities. Just knowing how to use a GUI is probably not enough to get a founded knowledge.

Whether you use Git on a command line from choice, as a preferred environment, or you need it because it is the only way to access the required functionality, there are a few shell features that Git can tap into to make your experience a lot friendlier.

Git-aware command prompt

It's useful to customize your **shell prompt** to show information about the state of the Git repository we are in.

Note

Shell prompt is a short text message that is written to the terminal or the console output to notify the user of the interactive shell that some typed

input is expected (usually a shell command).

This information can be as simple or as complex as you want. Git's prompt might be similar to the ordinary command-line prompt (to reduce dissonance), or visibly different (to be able to easily distinguish that we are inside the Git repository).

There is an example implementation for `bash` and `zsh` shells in the `contrib/` area. If you install Git from the sources, just copy the `contrib/completion/git-prompt.sh` file to your home directory; if you have installed Git on Linux via a package manager, you will probably have it at `/etc/bash_completion.d/git-prompt.sh`. This file provides the `__git_ps1` shell function to generate a Git-aware prompt in the Git repositories, but first you need to source this file in your `.bashrc` or `.zshrc`:

```
if [ -f /etc/bash_completion.d/git-prompt.sh ]; then
    source /etc/bash_completion.d/git-prompt.sh
fi
```

The shell prompt is configured using environment variables. To set up prompt, you must change directly or indirectly the `PS1` (*prompt string one*, the default interaction prompt) environment variable. Thus, one solution to create a Git-aware command prompt is to include a call to the `__git_ps1` shell function in the `PS1` environment variable, by using command substitution:

```
export PS1='\u@\h:\w$\(__git_ps1 " (%s)") \$ '
```

Note that, for `zsh`, you would also need to turn on the command substitution in the shell prompt with `setopt PROMPT_SUBST` command.

Alternatively, for a slightly faster prompt and with a possibility of color, you can use `__git_ps1` to set `PS1`. This is done with the `PROMPT_COMMAND` environment variable in `bash` and with the `precmd()` function in `zsh`. You can find more information about this option in comments in the `git-prompt.sh` file; for `bash`, it could be:

```
PROMPT_COMMAND='__git_ps1 "\u@\h:\w""\\\$ "" (%s)"'
```

With this configuration (either solution), the prompt will look as follows:

```
bob@host.company.org:~/random/src (master) $
```

The `bash` and `zsh` shell prompts can be customized with the use of special characters, which get expanded by a shell. In the example used here, `\u` means the current user (`bob`), `\h` is the current hostname (`host.company.org`), `\w` means the current working directory (`~/random/src`), while `\$` prints the `$` part of the prompt (# if you are logged in as the root user). `\$(...)` in the `PS1` setup is used to call external commands and shell functions. `__git_ps1 " (%s)"` here calls the `__git_ps1` shell function provided by `git-prompt.sh` with a formatting argument: the `%s` token is the place-holder for the presented Git status. Note that you need to either use single quotes while setting the `PS1` variable from the command line, as in the example shown here, or escape shell substitution, so it is expanded while showing the prompt and not while defining the variable.

If you are using the `__git_ps1` function, Git will also display information about the current ongoing multistep operation: merging, rebasing, bisecting, and so on. For example, during an interactive rebase (`-i`) on the branch `master`, the relevant part of the prompt would be `master|REBASE-i`. It is very useful to have this information right here in the command prompt, especially if you get interrupted in the middle of operation.

It is also possible to indicate in the command prompt the state of the working tree, the index, and so on. We can enable these features by exporting the selected subset of these environment variables (for some features you can additionally turn it off on per-repository basis with provided boolean-valued configuration variables):

Variable/Configuration	Values	Effect
<code>GIT_PS1_SHOWDIRTYSTATE</code> <code>bash.showDirtyState</code>	Nonempty	This shows * for the unstaged changes and + for the staged changes.
<code>GIT_PS1_SHOWSTASHSTATE</code>	Nonempty	This shows \$ if something is stashed.

GIT_PS1_SHOWUNTRACKEDFILES bash.showUntrackedFiles	Nonempty	This shows % if there are untracked files in workdir.
GIT_PS1_SHOWUPSTREAM bash.showUpstream	Space-separated list of values: <ul style="list-style-type: none">• verbose• name• legacy• git• svn	This autoshows whether you are behind <, up to date "=", or ahead > of the upstream.name shows the upstream name and verbose shows the number of commits ahead/behind (with a sign). git compares HEAD to @{upstream} and svn to SVN upstream.
GIT_PS1_DESCRIBE_STYLE	One of values: <ul style="list-style-type: none">• contains• branch• describe• default	This provides extra information when on detached HEAD. contains uses newer annotated tags, branch newer tag or branch, describe uses older annotated tags, default shows if there is exactly matching tag.
GIT_PS1_SHOWCOLORHINTS (prompt command / precmd only)	Nonempty	Colored hint about the current dirty state and so on.
GIT_PS1_HIDE_IF_PWD_IGNORED bash.hideIfPwdIgnored	Nonempty	Does not show a Git-aware prompt if the current directory is set to be ignored by Git.

If you are using the `zsh` shell, you can take a look at the `zsh-git` set of scripts, the `zshkit` configuration scripts, or the `oh-my-zsh` framework available for `zsh`, instead of using `bash`—first completion and prompt setup from the Git contrib/. Alternatively you can use the `vcs_info` subsystem built-in into `zsh`.

Well, there are alternate prompt solutions also for `bash`, for example `git-radar`.

Note

You can, of course, generate your own Git-aware prompt. For example, you might want to split the current directory into the repository path part and the project subdirectory path part with the help of the `git rev-`

parse command.

Command-line completion for Git

Another shell feature that makes it easier to work with command-line Git is the programmable command-line completion. This feature can dramatically speed up typing Git commands. Command-line completion allows you to type the first few characters of a command, or a filename, and press the completion key (usually *Tab*) to fill the rest of the item. With the Git-aware completion, you can also fill in subcommands, command-line parameters, remotes, branches, and tags (ref names), each only where appropriate (for example, remote names are completed only if the command expects the remote name at a given position).

Git comes with built-in (but not always installed) support for the auto-completion of Git commands for the `bash` and `zsh` shells.

For `bash`, if the completion is not installed with Git (at `/etc/bash_completion.d/git.sh` in Linux by default), you need to get a copy of the `contrib/completion/git-completion.bash` file out of the Git source code. Copy it somewhere accessible, like your home directory, and source it from your `.bashrc` or `.bash_profile`:

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

Once the completion for Git is enabled, to test it you can type for example:

```
$ git check<TAB>
```

With Git completion enabled `bash` (or `zsh`) would autocomplete this to `git checkout`.

Similarly, in an ambiguous case, double *Tab* shows all the possible completions (though it is not true for all the shells):

```
$ git che<TAB><TAB>
checkout      cherry      cherry-pick
```

The completion feature also works with options; this is quite useful if you don't remember the exact option but only the prefix:

```
$ git config --<TAB><TAB>
--add           --get-regexp      --remove-section   --
unset
--file=         --global          --rename-section   --
unset-all
--get           --list            --replace-all
--get-all       --local           --system
```

Instead of the list of possible completions, some shells use (or can be configured to use) rotating completion, where with multiple possible completions, each *Tab* shows a different completion for the same prefix (cycling through them).

Note that the command-line completion (also called **tab completion**) generally works only in the interactive mode, and is based on the unambiguous prefix, not on the unambiguous abbreviation.

Autocorrection for Git commands

An unrelated, but similar to **tab completion**, built-in Git tool is **autocorrection**. By default, if you type something that looks like a mistyped command, Git helpfully tries to figure out what you meant. It still refuses to do it, even if there is only one candidate:

```
$ git chekout
git: 'chekout' is not a git command. See 'git --help'.
```

```
Did you mean this?
    checkout
```

However, with the `help.autoCorrect` configuration variable set to a positive number, Git will automatically correct and execute the mistyped commands after waiting for the given number of deciseconds (0.1 of second). You can use a negative value of this option for immediate execution, or zero to go back to default:

```
$ git chekout
WARNING: You called a Git command named 'chekout', which does
```

```
not exist.  
Continuing under the assumption that you meant 'checkout'  
in 0.1 seconds automatically...  
Your branch is up-to-date with 'origin/master'.
```

If there is more than one command that can be deduced from the entered text, nothing will be executed. This mechanism works only for Git commands; you cannot autocorrect subcommands, parameters, and options (as opposed to tab completion).

Making the command line prettier

Git fully supports a colored terminal output, which greatly aids in visually parsing the command output. A number of options can help you set the coloring to your preference.

First, you can specify when to use colors and for output of which commands. There is a `color.ui` master switch to control output coloring to turn off all the Git's colored terminal outputs and set them to `false`. The default setting for this configuration variable is `auto`, which makes Git color the output when it's going straight to a terminal, but omit the color-control codes when the output is redirected to a file or a pipe.

You can also set `color.ui` to `always`, though you'd rarely want this: if you want color codes in your redirected output, simply pass a `--color` flag to the Git command; conversely, the `--no-color` option would turn off colored output.

If you want to be more specific about which commands are colored and which parts of the output are colored, Git provides appropriate coloring settings: `color.branch`, `color.diff`, `color.interactive`, `color.status`, and so on. Like with the master switch `color.ui`, each of these can be set to `true`, `false`, `auto`, and `always`.

In addition, each of these settings has subsettings that you can use to set specific colors for specific parts of the output. The `color` value of such configuration variables, for example, `color.diff.meta` (to configure the coloring of meta information in your `diff` output), consists of space-

separated names of the foreground color, the background color (if set), and the text attribute.

You can set the color to any of the following values: `normal`, `black`, `red`, `green`, `yellow`, `blue`, `magenta`, `cyan`, or `white`. As for the attributes, you can choose from `bold`, `dim`, `ul` (underline), `blink`, and `reverse` (swap the foreground color with the background one).

The pretty formats for `git log` also include an option to set colors; see the `git log` documentation.

Alternative command line

To understand some of the rough edges of the Git user's interface, you need to remember that Git was developed to a large extent in the bottom-up fashion. Historically, Git began as a tool to write version control systems (you can see how early Git was used in the *A Git core tutorial for developers* documentation available at <https://www.kernel.org/pub/software/scm/git/docs/gitcore-tutorial.html> or <https://git-scm.com/docs/gitcore-tutorial>).

The first alternative "porcelain" for Git (alternative user interface) was Cogito. Nowadays, Cogito is no more; all of its features are long incorporated into Git (or replaced by better solutions). There were some attempts to write wrapper scripts (alternative UIs) designed to make it easy to learn and use, for example, **Easy Git (eg)**.

There are also external Git porcelains that do not intend to replace the whole user interface, but either provide access to some extra feature, or wrap Git to provide some restricted feature set. Patch management interfaces, such as **StGit**, **TopGit**, or **Guilt** (formerly **Git Queues (gq)**), are created to make it easy to rewrite, manipulate, and clean up selected parts of the unpublished history; these were mentioned as an alternative to an interactive rebase in [Chapter 8, Keeping History Clean](#). Then, there are single-file version control systems, such as Zit and SRC, which use Git as a backend.

Note

Besides alternative user interfaces, there are also different implementations of Git (defined as reading and writing Git repositories). They are at different stages of completeness. Besides core C implementation, there is JGit in Java, and also the `libgit2` project—the modern basis of Git bindings for various programming languages.

Graphical interfaces

You have learned how to use Git on the command line. The previous section told you how to customize and configure it to make it even more effective. But the terminal is not the end. There are other kinds of environments you can use to manage Git repositories. Sometimes, a visual representation is what you need.

Now, we'll take a short look at the various kinds of user-centered graphical tools for Git; the tour of Git administrative tools is left for the next chapter, [Chapter 11, *Git Administration*](#).

Types of graphical tools

Different tools and interfaces are tailored for different workflows. Some tools expose only a selected subset of the Git functionality, or encourage a specific way of working with version control.

To be able to make an informed choice selecting a graphical tool for Git, you need to know what types of operations the different types of tools do support. Note that one tool can support more than one type of uses.

First there is a **graphical history viewer**. You can think of it as a powerful GUI over `git log`. This is the tool to be used when you are trying to find something that happened in the past, or you are visualizing and browsing your project's history and the layout of branches. Such tools usually accept revision selection command-line options, such as `--all`. Command-line Git has `git log --graph` and less used `git show-branch` that use ASCII-art to show the history.

A similar tool is **graphical blame**, showing the line-wise history of a file. For each line, it can show when that line was created and when it was moved or copied to the current place. You can examine the details of each of the commits shown, and usually browse through the history of the lines in a file. Other tools with similar applications, namely examining the evolution of the line range (`git log -L`) and the so called

pickaxe search (`git log -s`) does not have many GUIs.

Next, there are **commit tools** meant primarily to craft (and amend) commits, though usually they also include some kind of worktree management (for example ignoring files and switching branches) and **management of remotes**. Such tools would usually show unstaged and staged changes, allowing you to move files between these states. Some of those tools even allow to stage and unstage individual chunks of changes, like interactive versions of `git add`, `git reset`, and so on. A graphical version of an interactive add is described in [Chapter 4, Managing Your Worktree](#), and mentioned in [Chapter 3, Developing with Git](#).

Then, we have **file manager integration** (or **graphical shell integration**). These plugins usually show the status of the file in Git (tracked/untracked/ignored) using icon overlays. They can offer a context menu for a repository, directory, and file, often with accompanying keyboard shortcuts. They may also bring drag and drop support.

Programmer editors and integrated development environments (IDEs) often offer support for **IDE integration** with Git (or version control in general). These offer repository management (as a part of team project management), make it possible to perform Git operations directly from the IDE, show the status of the current file and the repository, and perhaps even annotate the view of the file with version control information. They often include the commit tool, remote management, the history viewer, and the diff viewer.

Git repository's hosting sites often offer workflow-oriented **desktop clients**. These mostly focus on a curated set of commonly used features that work well together in the flow. They automate common Git tasks. They are often designed to highlight their service, offering extra features and integration, but they will work with any repository hosted anywhere.

Graphical diff and merge tools

Graphical diff tools and graphical merge tools are somewhat special case. In these categories, Git includes the commands for integration with third-party graphical tools, namely, `git difftool` and `git mergetool`. These tools would then be called from the Git repository. Note that this is different from the external diff or diff merge drivers, which replace ordinary `git diff` or augment it.

Although Git has an internal implementation of diff and a mechanism for merge conflict resolutions (see [Chapter 7, Merging Changes Together](#)), you can use an external graphical diff tool instead. These are often used to show the differences better (usually, as a side-by-side diff, possibly with refinements), and help resolve a merge (often with a three-pane interface).

Configuring the graphical `diff` tool, or the graphical `merge` tool, takes a number of custom settings. To tell which tool to use for `diff` and `merge`, respectively, you can set up `diff.tool` and `merge.tool`, respectively . Without setting for example "`merge.tool`" the "`git mergetool`" command would print the information on how to configure it, and will attempt to run one of predefined tools:

```
$ git mergetool
```

This message is displayed because '`merge.tool`' is not configured.

See '`git mergetool --tool-help`' or '`git help config`' for more details.

'`git mergetool`' will now attempt to use one of the following tools:

tortoisemerge emerge vimdiff

No files need merging

Running `git mergetool --tool-help` will show all the available tools, including those that are not installed. In case the tool you use is not in `$PATH`, or it has a wrong version of the tool, you can use `mergetool <tool>.path` to set or override the path for the given tool:

```
$ git mergetool --tool-help  
'git mergetool --tool=<tool>' may be set to one of the  
following:
```

```
vimdiff
```

```
[...]
```

The following tools are valid, but not currently available:

```
araxis
```

```
[...]
```

Some of the tools listed above only work in a windowed environment. If run in a terminal-only session, they will fail.

If there is no built-in support for your tool, you can still use it; you just need to configure it. The `mergetool.<tool>.cmd` configuration variable specifies how to run the command, while `mergetool`.

`<tool>.trustExitCode` tells Git whether the exit code of that program indicates a successful merge resolution or not. The relevant fragment of the configuration file (for a graphical merge tool named `extMerge`) could look as follows:

```
[merge]
  tool = extMerge
[mergetool "extMerge"]
  cmd = extMerge "$BASE" "$LOCAL" "$REMOTE" "$MERGED"
```

Graphical interface examples

In this section, you will be presented with a selection of tools around Git that you could use, or that might prompt you to research further. A nice way to start such a research is to list some selected GUI clients.

There are two visual tools that are a part of Git and are usually installed with it, namely `gitk` and `git-gui`. They are written in Tcl/Tk. `Gitk` is a graphical history viewer, while `git gui` is a commit tool; there is also `git gui blame`, a visually interactive line-history browser. These tools are interconnected, for example, browsing history from `git gui` opens `gitk`.

Visual tools do not need to use the graphical environment. There is `tig` (Text Interface for Git), aurses-based text-mode interface, which functions as a repository browser and a commit tool, and can act as a Git pager.

There is `git cola` developed in Python and available for all the operating systems, which includes commit tools and remotes management, and also a diff viewer. Then, there is the simple and colorful `Gitg` tool for GNOME; you will get a graphical history viewer, diff viewer, and file browser.

One of the more popular open-source GUI tools for MacOS is `Gitx`. There are a lot of forks of this tool; one of the more interesting ones is `Gitbox`. It features both the history viewer and commit tools.

For MS Windows, there is `TortoiseGit` and `git-cheetah`, both of which offer integration into the Windows context menu, so you can perform Git commands inside Windows Explorer (the file manager integration and shell interface).

Both GitHub Inc. and Atlassian released a desktop GUI tool that you can easily use with your GitHub or Bitbucket repository, respectively, but it is not limited to it. Both `GitHub Client` and `SourceTree` feature repository management in addition to other common facilities.

Configuring Git

So far, while describing how Git works and how to use it, we have introduced a number of ways to change its behavior. Here, it will be explained in the systematic fashion how to configure Git operations on a temporary and permanent basis. We will also see how you can make Git behave in a customized fashion by introducing and reintroducing several important configuration settings. With these tools, it's easy to get Git to work the way you want it to.

Command-line options and environment variables

Git processes the switches that change its behavior in a hierarchical fashion, from the least specific to the most specific one, with the most specific one (and shortest term) moved earlier taking precedence.

The most specific one, overriding all the others, is the command-line options. They affect, obviously, only the current Git command.

Note

One issue to note is that some command-line options, for example `--no-pager` or `--no-replace-objects` go to the git wrapper, not to the Git command itself. Examine, for example, the following line to see the distinction:

```
$ git --no-replace-objects log -5 --oneline --graph --decorate
```

You can find the conventions used through the Git command-line interface on <https://www.kernel.org/pub/software/scm/git/docs/gitcli.html> manpage.

The second way to change how the Git command works is to use environment variables. They are specific to the current shell, and you need to use `export` to propagate the variables to the subprocesses if replacement is used. There are some environment variables that apply to

all core Git commands, and some that are specific to a given (sub)command.

Git also makes use of some nonspecific environment variables. These are meant as a last resort; they are overridden by their Git specific equivalents. Examples include variables such as `PAGER` and `EDITOR`.

Git configuration files

The final way to customize how Git works is with the configuration files. In many cases, there is a command-line option to configure an action, an environment variable for it, and finally a configuration variable, in the descending order of preference.

Git uses a series of configuration files to determine nondefault behavior that you might want to have. There are three layers of those that Git looks for configuration values. Git reads all these files in order from the least specific to the most specific one. The settings in the later ones override those set in the earlier ones. You can access the Git configuration with the `git config` command: by default, it operates on the union of all the files, but you can specify which one you want to access with the command-line options. You can also access any given file following the configuration file syntax (such as the `.gitmodules` file mentioned in [Chapter 9, Managing Subprojects - Building a Living Framework](#)) by using the `--file=<pathname>` option (or the `GIT_CONFIG` environment variable).

Note

You can also read the values from any blob with configuration-like contents; for example, you may use `git config--blob=master:.gitmodules` to read from the `.gitmodules` file in the `master` branch.

The first place Git looks for configuration is the system-wide configuration file. If Git is installed with the default settings, it can be found in `/etc/gitconfig`. Well, at least, on Linux it is there, as the

Filesystem Hierarchy Standard (FHS) states that `/etc` is the directory for storing the host-specific system-wide configuration files; Git for Windows puts this file in the subdirectory of its Program Files folder. This file contains the values for every user on the system and all their repositories. To make `git config` read and write from and to this file specifically (and to open it with `--edit`), pass the `--system` option to the `git config` command.

You can skip the reading settings from this file with the `GIT_CONFIG_NOSYSTEM` environment variable. This can be used to set up a predictable environment or to avoid using a buggy configuration you can't fix.

The next place Git looks is `~/.gitconfig`, falling back to `~/.config/git/config` if it exists (with the default configuration). This file is specific to each user and it affects all the user's repositories. If you pass the option `--global` to `git config`, it would read and write from this file specifically. Reminder: here, as in the other places, `~` (the tilde character) denotes the home directory of the current user (`$HOME`).

Finally, Git looks for the configuration values in the per-repository configuration file in the Git repository you are currently using, which is (by default and for nonbare repositories) `.git/config`. Values set there are specific to that local single repository. You can make Git read and write to this file by passing the `--local` option.

Each of these levels (system, global, and local) overrides values in the previous level, so for example, values in `.git/config` trump those in `~/.gitconfig`; well, unless the configuration variable is multivalued.

Note

You can use this to have your default identity in the per-user file and to override it if necessary on a per-repository basis with a per-repository configuration file.

The syntax of Git configuration files

Git's configuration files are plain text, so you can also customize Git's behavior by manually editing the chosen file. The syntax is fairly flexible and permissive; whitespaces are mostly ignored (contrary to `.gitattributes`). The hash `#` and semicolon `;` characters begin comments, which last until the end of the line. Blank lines are ignored.

The file consists of sections and variables, and its syntax is similar to the syntax of INI files. Both the section names and variable names are case-insensitive. A section begins with the name of the section in square brackets `[section]` and continues until the next section. Each variable must begin at some section, which means that there must be a section header before the first setting of a variable. Sections can repeat and can be empty.

Sections can be further divided into subsections. Subsection names are case-sensitive and can contain any character except newline (double quotes `"` and backslash `\` must be escaped as `\"` and `\\`, respectively). The beginning of the subsection will look as follows:

```
[section "subsection"]
```

All the other lines (and the remainder of the line after the section header) are recognized as a setting variable in the `name = value` form. As a special case, just `name` is a shorthand for `name = true` (boolean variables). Such lines can be continued to the next line by ending it with `"\"` (the backslash character), that is by escaping the end-of-line character. Leading and trailing whitespaces are discarded; internal whitespaces within the value are retained verbatim. You can use double quotes to preserve leading or trailing whitespaces in values.

You can include one `config` file from another by setting the special variable `include.path` to the path of the file to be included. The included file will be expanded immediately, similar to the mechanism of `#include` in C and C++. The path is relative to the configuration file with the `include` directive. You can turn this feature off with `--no-includes`.

Tip

Types of configuration variables and type specifiers

While requesting (or writing) a `config` variable, you may give a *type specifier*. It can be `--bool`, which ensures that the returned value is `true` or `false`; `--int`, which expands the optional value suffix of `k` (1024 elements), `m` (1024k), or `g` (1024m); `--path`, which expands `~` for the value of `$HOME`; and `~user` for the home directory of the given user. There is also `--bool-or-int` and a few options related to storing colors and retrieving color escape codes; see the `git config` documentation.

Accessing the Git configuration

You can use the `git config` command to access the Git configuration, starting from listing the configuration entries in a canonical form, through examining individual variables, to editing and adding entries.

You can query the existing configuration with `git config --list`, adding an appropriate parameter if you want to limit to a single configuration layer. On a Linux box with the default installation, in the fresh empty Git repository just after `git init`, the local (per-repository) setting would look approximately like the following:

```
$ git config --list --local
core.repositoryformatversion=0
core.filemode=false
core.bare=false
core.logallrefupdates=true
```

You can also query a single key with `git config`, limiting or not the scope to the specified file, by giving the name of configuration variable as a parameter (optionally preceded by `--get`), with the section, optional subsection, and variable name (key) separated by dot:

```
$ git config user.email
alice@example.com
```

This would return the last value, that is, the one with the most precedence. You can get all the values with `--get-all`, or specific keys with `--get-regexp=<match>`. This is quite useful while accessing a

multivalued option like `refsspecs` for a remote.

With `--get`, `--get-all`, and `--get-regexp`, you can also limit the listing (and the settings for multiple-valued variables) to only those variables matching the value `regexp` (which is passed as an optional last parameter). For example:

```
$ git config --get core.gitproxy 'for kernel\.org$'
```

You can also use the `git config` command to set the configuration variable value. For example, to set the e-mail address of the user, which is to be common to most of his or her repositories, you can run the following:

```
$ git config --global user.name "Alice Developer"
```

To change multivar, you can use:

```
$ git config core.gitproxy '"ssh" for kernel.org' 'for kernel\.org$'
```

The local layer (per-repository file) is the default for writing, if nothing else is specified. For multivalue configuration options, you can add multiple lines to it by using the `--add` option.

It is also very easy to delete configuration entries with `git config --unset`.

Instead of setting all the configuration values on the command line, as shown in the preceding example, it is possible to set or change them just by editing the relevant configuration file directly. Simply open the configuration file in your favorite editor, or run the `git config --edit` command.

The local repository configuration file just after a fresh init on Linux looks as follows:

```
[core]
repositoryformatversion = 0
filemode = true
```

```
bare = false  
logallrefupdates = true
```

Basic client-side configuration

You can divide the configuration options recognized by Git into two categories: client-side and server-side. The majority of the options are about configuring your personal working preferences; they are client-side. The server-side configuration will be touched upon in more detail in [Chapter 11, *Git Administration*](#); here you will find only basics.

There are many supported configuration options, but only a small fraction of them needs to be set; a large fraction of them has sensible defaults, and explicitly setting them is only useful in certain edge cases. There are a lot of options available; you can see a list of all the options with `git config --help`. Here we'll be covering only the most common and most useful options.

Two variables that really need to be set up are `user.email` and `user.name`. Those configuration variables define the author's identity. Also, if you are signing annotated tags or commits (as discussed in [Chapter 5, *Collaborative Development with Git*](#)), you might want to set up your GPG signing key ID. This is done with the `user.signingKey` configuration setting.

By default, Git uses whatever you've set on the system as your default text editor (defined with the `VISUAL` or `EDITOR` environment variables; the first only for the graphical desktop environment) to create and edit your commit and tag messages. It also uses whatever you have set as the pager (`PAGER`) for paginating and browsing the output of the Git commands. To change this default to something else, you can use the `core.editor` setting. The same goes for `core.pager`. Git would ultimately fall back on the `vi` editor and on the `less` pager.

Note

With Git, the pager is invoked automatically. The default `less` pager supports not only pagination, but also incremental search for example.

Also, with the default configuration (the `LESS` environment variable is not set) `less` invoked by Git works as if it was invoked with `LESS=FRX`. This means that it would skip pagination of there is less than one page of output, it would pass through ANSI color codes, and it would not clear screen on exit.

Creating commit messages is also affected by `commit.template`. If you set this configuration variable, Git will use that file as the default message when you commit. The template is not distributed with the repository in general. Note that Git would add the status information to the commit message template, unless it is forbidden to do it by setting `commit.status` to `false`.

Such a template is quite convenient if you have a commit-message policy, as it greatly increases the chances of this policy being followed. It can, for example, include the commented-out instructions for filling the commit message. You can augment this solution with an appropriate hook that checks whether the commit message matches the policy (see the *Commit process hooks* section in this chapter).

The status of the files in the working area is affected by the ignore patterns and the file attributes (see [Chapter 4, Managing Your Worktree](#)). You can put ignore patterns in your project's in-tree `.gitignore` file (usually tracked '`.gitignore`' is about which files are tracked, and is tracked itself by Git (not by itself). itself), or in the `.git/info/excludes` file for local and private patterns, to define which files are not interesting. These are project-specific; sometimes, you would want to write a kind of global (per-user) `.gitignore` file. You can use `core.excludesFile` to customize the path to the said file; in modern Git, there is a default value for this path, namely, `~/.config/git/ignore`. There is also a corresponding `core.attributesFile` for this kind of global `.gitattributes` files, which defaults to `~/.config/git/attributes`.

Note

Actually, it is `$XDG_CONFIG_HOME/git/ignore`; if the `$XDG_CONFIG_HOME`

environment variable is not set or is empty, `$HOME/.config/git/ignore` is used.

Although Git has an internal implementation of diff, you can set up an external tool to be used instead with the help of `diff.external`. You would usually want to create a wrapper script that massages the parameters that Git passes to it, and pass the ones needed in the order external diff requires. By default, Git passes the following arguments to the diff program:

```
path old-file old-hex old-mode new-file new-hex new-mode
```

See also the *Graphical diff and merge tools* section for the configuration of `git difftool` and `git mergetool`.

The rebase and merge setup, configuring pull

By default, when performing `git pull` (or equivalent), Git would use the `merge` operation to join the local history and the history fetched from the remote. This would create a merge commit if the history of the local branch has diverged from the remote one. Some argue that it is better to avoid all these merge commits and create mostly a linear history by using `rebase` instead (for example, with `git pull --rebase`) to join histories. You can find more information on this topic in [Chapter 7, Merging Changes Together](#).

There are several configuration settings that can be used to make the `git pull` default to rebase, to set up tracking, and so on. There is the `pull.rebase` configuration option and a branch-specific `branch.<name>.rebase` option that, when set to `true`, tells Git to perform rebase instead of merge during pull (for the `<name>` branch only in a later case). Both can also be set to `preserve` to run rebase with the `--preserve-merges` option, to have local merge commits not be flattened in the rebase.

You can make Git automatically set up the per-branch "pull to rebase" configuration while creating specific kinds of new branches with `branch.autoSetupRebase`. You can set it to `never`, `local` (for locally

tracked branches only), `remote` (for remote tracked branches only), or `always` (for local plus remote).

Preserving undo information – the expiry of objects

By default, Git will automatically remove unreferenced objects, clean reflogs of stale entries, and pack loose objects, all to keep the size of the repository down. You can also run the garbage collection manually with the `git gc` command. You should know about repository's object-oriented structure from [Chapter 8, *Keeping History Clean*](#).

Git will, for safety reasons, use a grace period of two weeks while removing unreferenced objects for; this can be changed with the `gc.pruneExpire` configuration: the setting is usually a relative date (for example, `1.month.ago`; you can use dots as a word separator). To disable the grace period (which is usually done from the command line), the value `now` can be used.

The branch tip history is kept for 90 days by default (or `gc.reflogExpire`, if set) for reachable revisions, and for 30 days (or `gc.reflogExpireUnreachable`) for reflog entries that are not a part of the current history. Both settings can be configured on a per-refname basis, by supplying a pattern of the ref name to be matched as a subsection name, that is, `gc.<pattern>.reflogExpire`, and similar for the other setting. This can be used to change the expire settings for `HEAD` or for `refs/stash` (see [Chapter 4, *Managing Your Worktree*](#)), or for remote-tracking branches `refs/remotes/*` separately. The setting is a length of time (for example, `6.months`); to completely turn off reflog expiring use the value of `never`. You can use the latter for example to switch off expiring of `stash` entries.

Formatting and whitespace

Code formatting and whitespace issues are some of the more frustrating and subtle problems you may encounter while collaborating, especially with cross-platform development. It's very easy for patches and merges to introduce subtle whitespace changes, because of editors silently introducing such changes (often not visible) and a different notion of

line endings on different operating systems: MS Windows, Linux, and MacOS X. Git has a few configuration options to help with these issues.

One important issue for cross-platform work is the notion of **line-ending**. This is because MS Windows uses a combination of a carriage return (CR) character and a linefeed (LF) character for new lines in text files, whereas MacOS and Linux use only a linefeed character. Many editors on MS Windows will silently replace existing LF-style line endings with CRLF or use CRLF for new lines, which leads to subtle but annoying troubles.

Git can handle this issue by auto-converting line endings into LF when you add a file to the index. If your editor uses CRLF line endings, Git can also convert line-endings to the native form when it checks out code in your filesystem. There are two configuration settings that affect this matter: `core.eol` and `core.autocrlf`. The first setting, `core.eol`, sets the line ending to be used while checking out files into the working directory for files that have the `text` property set (see the next section, *Per-file configuration with gitattributes*, which summarizes and recalls information about the file attributes from [Chapter 4, Managing Your Worktree](#)).

The second and older setting, `core.autocrlf`, can be used to turn on the automatic conversion of line endings to CRLF. Setting it to `true` converts the LF line endings in the repository into CRLF when you check out files, and vice versa when you stage them; this is the setting you would probably want on a Windows machine. (This is almost the same as setting the `text` attribute to `auto` on all the files and `core.eol` to `crlf`.) You can tell Git to convert CRLF to LF on a commit but not the other way around by setting `core.autocrlf` to `input` instead; this is the setting to use if you are on a Linux or Mac system. To turn off this functionality, recording the line-endings in the repository as they are set this configuration value to `false`.

This handles one part of the whitespace issues: line-ending variance, and one vector of introducing them: editing files. Git also comes with the way to detect and fix some of other whitespace issues. It can look for a

set of common whitespace problems to notice. The `core.whitespace` configuration setting can be used to activate them (for those disabled by default), or turn them off (for those enabled by default). The three that are turned on by default are:

- `blank-at-eol`: This looks for trailing spaces at the end of a line
- `blank-at-eof`: This notices blank lines at the end of a file
- `space-before-tab`: This looks for spaces immediately before the tabs at the initial (beginning) indent part of the line

The `trailing-space` value in `core.whitespace` is a shorthand to cover both `blank-at-eol` and `blank-at-eof`. The three that are disabled by default but can be turned on are:

- `indent-with-non-tab`: This treats the line that is indented with space characters instead of the equivalent tabs as an error (where equivalence is controlled by the `tabwidth` option); this option enforces indenting with *Tab* characters.
- `tab-in-indent`: This watches for tabs in the initial indentation portion of the line (here, `tabwidth` is used to fix such whitespace errors); this option enforces indenting with space characters.
- `cr-at-eol`: This tells Git that carriage returns at the end of the lines are OK (allowing CRLF endings in the repository).

You can tell Git which of these you want enabled or disabled by setting `core.whitespace` to the comma separated list of values. To disable an option, prepend it with the `"-"` prefix in front of the value. For example, if you want all but `cr-at-eol` and `tab-in-indent` to be set, and also while setting the tab space value to 4, you can use:

```
$ git config --local core.whitespace \
    trailing-space,space-before-tab,indent-with-non-
    tab,tabwidth=4
```

You can also set these options on a per-file basis with the `whitespace` attribute. For example, you can use it to turn off checking for whitespace problems in test cases to handle whitespace issues, or ensure that the Python 2 code indents with spaces:

```
*.py whitespace=tab-in-indent
```

Git will detect these issues when you run a `git diff` command and inform about them using the `color.diff.whitespace` color, so you can notice them and possibly fix them before you create a new commit. While applying patches with `git apply`, you can ask Git to either warn about the whitespace issues with `git apply --whitespace=warn`, error out with `--whitespace=error`, or you can have Git try to automatically fix the issue with `--whitespace=fix`. The same applies to the `git rebase` command as well.

Server-side configuration

There are a few configuration options available for the server-side of Git. They would be described in more detail in [Chapter 11, *Git Administration*](#); here you will find a short summary of some of the more interesting parameters.

You can make the Git server check for object consistency, namely, that every object received during a push matches its SHA-1 identifier and that points to a valid object with a `receive.fsckObjects` Boolean-valued configuration variable. It is turned off by default because `git fsck` is a fairly expensive operation, and it might slow down operation, especially on large pushes (which are common in large repositories). This is a check against faulty or malicious clients.

If you rewrite commits that you have already pushed to a server (which is bad practice, as explained in [Chapter 8, *Keeping History Clean*](#)) and try to push again, you'll be denied. The client might, however, force-update the remote branch with the `--force` flag to the `git push` command. However, the server can be told to refuse force-pushes by setting `receive.denyNonFastForward` to `true`.

The `receive.denyDeletes` setting blocks one of the workarounds to the `denyNonFastForward` policy, namely, deleting and recreating a branch. This forbids the deletion of branches and tags; you must remove refs from the server manually.

All of these features could also be implemented via the server-side receive-like hooks; this will be covered in the *Installing a Git hook* section, and also to some extent in [Chapter 11, Git Administration](#).

Per-file configuration with gitattributes

Some of the customizations can also be specified for a path (perhaps via glob) so that Git applies these settings only for a subset of files or for a subdirectory. These path-specific settings are called `gitattributes`.

The order of precedence of applying this type of settings starts with the per-repository local (per-user) per-path settings in the

`$GIT_DIR/info/attributes` file. Then, the `.gitattributes` files are consulted, starting with the one in the same directory as the path in question, going up through the `.gitattributes` files in the parent directories, up to the top level of the worktree (the root directory of a project). Finally, the global per-user attributes file (specified by `core.attributesFile`, or at `~/.config/git/attributes` if this is not set) and the system-wide file (in `/etc/gitattributes` in the default installation) are considered.

Available Git attributes are described in detail in [Chapter 4, Managing Your Worktree](#). Using attributes, you can, among others, do things such as specify the separate merge strategies via merge drivers for the specific kind of files (for example, `ChangeLog`), tell Git how to diff non-text files, or have Git filter content during checkout (on writing to the working area, that is, to the filesystem) and checkin (on staging contents and committing changes to the repository, that is, creating objects in the repository database).

Tip

Syntax of the Git attributes file

A `gitattributes` file is a simple text file that sets up the local configuration on a per-path basis. Blank lines, or lines starting with the hash character (#) are ignored; thus, a line starting with # serves as a comment, while

blank lines can serve as separators for readability. To specify a set of attributes for a path, put a pattern followed by an attributes list, separated by a horizontal whitespace:

```
pattern attribute1 attribute2
```

When more than one pattern matches the path, a later line overrides an earlier line, just like for the `.gitignore` files (you can also think that the Git attributes files are read from the least specific system-wide file to the most specific local repository file).

Git uses a backslash (\) as an escape character for patterns. Thus, for patterns that begin with a hash, you need to put a backslash in front of the first hash (that is written as \#). Because the attributes information is separated by whitespaces, trailing spaces in the pattern are ignored and inner spaces are treated as end of pattern, unless they are quoted with a backslash (that is, written as "\ ").

If the pattern does not contain a slash (/), which is a directory separator, Git will treat the pattern as a shell glob pattern and will check for a match against the pathname relative to the location of the `.gitattributes` file (or top level for other attribute files). Thus, for example, the `*.c` patterns match the C files anywhere down from the place the `.gitattributes` file resides. A leading slash matches the beginning of the pathname. For example, `/*.c` matches `bisect.c` but not `builtin/bisect--helper.c`, while `*.c` pattern would match both.

If the pattern includes at least one slash, Git will treat it as a shell glob suitable for consumption by the `fnmatch(3)` function call with the `FNM_PATHNAME` flag. This means that the wildcards in the pattern will not match the directory separator, that is, the slash (/) in the pathname; the match is anchored to beginning of the path. For example, the `include/*.h` pattern matches `include/version.h` but not `include/linux/asm.h` or `libxdiff/includes/xdiff.h`. Shell glob wildcards are: * matching any string (including empty), ? matching any single character, and the [...] expression matching the character class (inside brackets, asterisks and question marks lose their special

meaning); note that unlike in regular expressions, the complementation/negation of character class is done with `!` and not `^`. For example to match anything but a number one can use `[!0-9]` shell pattern, which is equivalent to `[^0-9]` regexp.

Two consecutive asterisks (`**`) in patterns may have a special meaning, but only between two slashes (`/**/`), or between a slash and at the beginning or the end of pattern. Such a wildcard matches zero or more path components. Thus, a leading `**` followed by a slash means match in all directories, while trailing `/**` matches every file or directory inside the specified directory.

Each attribute can be in one of the four states for a given path. First, it can be *set* (the attribute has special value of `true`); this is specified by simply listing the name of the attribute in the attribute list, for example, `text`. Second, it can be *unset* (the attribute has a special value of `false`); this is specified by listing the name of the attribute prefixed with minus, for example, `-text`. Third, it can be set to a specific value; this is specified by listing the name of the attribute followed by an equal sign and its value, for example, `-text=auto` (note that there cannot be any whitespace around the equal sign as opposed to the configuration file syntax). If no pattern matches the path, and nothing says if the path has or does not have attributes, the attribute is said to be unspecified (you can override a setting for the attribute, forcing it to be explicitly unspecified with `!text`).

If you find yourself using the same set of attributes over and over for many different patterns, you should consider defining a macro attribute. It can be defined in the local, or global, or system-wide attributes file, but only in the top level `.gitattributes` file. The macro is defined using `[attr]<macro>` in place of the file pattern; the attributes list defines the expansion of the macro. For example, the built-in `binary` macro attribute is defined as if using:

```
[attr]binary -diff -merge -text
```

Automating Git with hooks

There are usually certain prerequisites to the code that is produced, either self-induced or enforced externally. The code should be always able to compile and pass at least a fast subset of the tests. With some development workflows, each commit message may need to reference an issue ID (or fit message template), or include a digital certificate of origin in the form of the `Signed-off-by` line. In many cases, these parts of the development process can be automated by Git.

Like many programming tools, Git includes a way to fire custom functionality contained in the user-provided code (custom scripts), when certain important pre-defined actions occur, that is, when certain events trigger. Such a functionality invoked as a event handler is called a **hook**. It allows to take an additional action and, at least for some hooks, also to stop the triggered functionality.

Hooks in Git can be divided into the client-side and the server-side hooks. **Client-side hooks** are triggered by local operations (on client) such as committing, applying a patch series, rebasing, and merging. **Server-side hooks** on the other hand run on the server when the network operations such as receiving pushed commits occur.

You can also divide hooks into pre hooks and post-hooks. **Pre hooks** are called before an operation is finished, usually before the next step while performing an operation. If they exit with a nonzero value, they will cancel the current Git operation. **Post hooks** are invoked after an operation finishes and can be used for notification and logs; they cannot cancel an operation.

Installing a Git hook

The hooks in Git are executable programs (usually scripts), which are stored in the `hooks/` subdirectory of the Git repository administrative area, that is in `.git/hooks/` for non-bare repositories. Hook programs are named each after an event that triggers it; this means that if you

want for one event to trigger more than one script, you will need to implement multiplexing yourself.

When you initialize a new repository with `git init` (this is done also while using `git clone` to create a copy of the other repository; `clone` calls `init` internally), Git populates the `hooks` directory with a bunch of inactive example scripts. Many of these are useful by themselves, but they also document the hook's API. All the examples are written as shell or Perl scripts, but any properly named executable would work just fine. If you want to use bundled example hook scripts, you'll need to rename them, stripping the `.sample` extension and ensuring that they have the executable permission bit.

A template for repositories

Sometimes you would want to have the same set of hooks for all your repositories. You can have a global (per-user and system-wide) configuration file, a global attributes file, and a global ignore list. It turns out that it is possible to select hooks to be populated during the creation of the repository. The default sample hooks that get copied to the `.git/hooks` repository are populated from `/usr/share/git-core/templates`.

Also, the alternative directory with the repository creation templates can be given as a parameter to the `--template` command-line option (to `git clone` and `git init`), as the `GIT_TEMPLATE_DIR` environment variable, or as the `init.templateDir` configuration option (which can be set in a per-user configuration file). This directory must follow the directory structure of `.git` (of `$GIT_DIR`), which means that the hooks need to be in the `hooks/` subdirectory there.

Note, however, that this mechanism has some limitations. As the files from the template directory are only copied to the Git repositories on their initialization, updates to the template directory do not affect the existing repositories. Though you can re-run `git init` in the existing repository to reinitialize it, just remember to save any modifications made to the hooks.

Note

Maintaining hooks for a team of developers can be tricky. One possible solution is to store your hooks in the actual project directory (inside project working area), or in a separate hooks repository, and create a symbolic link in `.git/hooks`, as needed.

There are even tools and frameworks for Git hook management; you can find examples of such tools listed on <http://githooks.com/>.

Client-side hooks

There are quite a few client-side hooks. They can be divided into the commit-workflow hooks (a set of hooks invoked by the different stages of creating a new commit), apply-email workflow hooks, and everything else (not organized into a multihook workflow).

Note

It is important to note that hooks are *not* copied when you clone a repository. This is done partially for security reasons, as hooks run unattended and mostly invisible. You need to copy (and rename) files themselves, though you can control which hooks get installed while creating or reinitializing a repository (see the previous subsection). This means that you cannot rely on the client-side hooks to enforce a policy; if you need to introduce some hard requirements, you'll need to do it on the server-side.

Commit process hooks

There are four client-side hooks invoked (by default) while committing changes. They are as follows:

1. The `pre-commit` hook is run first, even before you invoke the editor to type in the commit message. It is used to inspect the snapshot to be committed, to see whether you haven't forgotten anything. A nonzero exit from this hook aborts the commit. You can bypass invoking this hook altogether with `git commit --no-verify`. This

hook takes no parameters.

This hook can, among others, be used to check for the correct code style, run the static code analyzer (linter) to check for problematic constructs, make sure that the code compiles and that it passes all the tests (and that the new code is covered by the tests), or check for the appropriate documentation on a new functionality. The default hook checks for whitespace errors (trailing whitespace by default) with `git diff --check` (or rather its plumbing equivalent), and optionally for non-ASCII filenames in the changed files. You can, for example, make a hook that asks for a confirmation while committing with a dirty work-area (for the changes in the worktree that would not be a part of the commit being created); though it is an advanced technique. Or, you can try to have it check whether there are documentations and unit tests on the new methods.

2. The `prepare-commit-msg` hook is run after the default commit message is created (including the static text of the file given by `commit.template`, if any), and before the commit message is opened in the editor. It lets you edit the default commit message or create a template programmatically, before the commit author sees it. If the hook fails with a nonzero status, the commit will be aborted. This hook takes as parameters the path to the file that holds the commit message (later passed to the editor) and the information about source of the commit message (the latter is not present for ordinary `git commit`): `message` if the `-m` or `-F` option was given, `template` if the `-t` option was given or `commit.template` was set, `merge` if the commit is merged or the `.git/MERGE_MSG` file exists, `squash` if the `.git/SQUASH_MSG` file exists, or `commit` if the message comes from the other commit: the `-c`, `-C`, or `--amend` option was given. In the last case, the hook gets additional parameters, namely, a SHA-1 of the commit that is the source of the message.

The purpose of this hook is to edit or create the commit message, and this hook is not suppressed by the `--no-verify` option. This hook is most useful when it is used to affect commits where the default message is autogenerated, such as the templated commit

message, merged commits, squashed commits, and amended commits. The sample hook that Git provides comments out the `Conflict:` part of the merge commit message.

Another example of what this hook can do is to use the description of the current branch given by `branch.<branch-name>.description`, if it exists, as a base for a branch-dependent dynamic commit template. Or perhaps, check whether we are on the topic branch, and then list all the issues assigned to you on a project issue tracker, to make it easy to add the proper artefact ID to the commit message.

3. The `commit-msg` hook is run after the developer writes the commit message, but before the commit is actually written to the repository. It takes one parameter, a path to the temporary file with the commit message provided by user (by default `.git/COMMIT_EDITMSG`).

If this script exits with a nonzero status, Git aborts the commit process, so you can use it to validate that, for example, the commit message matches the project state, or that the commit message conforms to the required pattern. The sample hook provided by Git can check, sort, and remove duplicated `Signed-off-by:` lines (which might be not what you want to use, if signoffs are to be a chain of provenance). You could conceivably check in this hook whether the references to the issue numbers are correct (and perhaps expand them, adding the current summary of each mentioned issue).

Gerrit Code Review provides a `commit-msg` hook (which needs to be installed in the local Git repository) to automatically create, insert, and maintain a unique `Change-Id:` line above the signoffs during `git commit`. This line is used to track the iterations of coming up with a commit; if the commit message in the revision pushed to Gerrit lacks such information, the server will provide instructions on how to get and install that hook script.

4. The `post-commit` hook runs after the entire process is completed. It doesn't take any parameters, but at this point of the commit operation the revision that got created during commit is available as `HEAD`. The exit status of this hook is ignored.

Generally, this script (like most of the `post-*` scripts) is most often used for notifications and logging, and it obviously cannot affect the outcome of `git commit`. You can use it, for example, to trigger a local build in a continuous integration tool such as Jenkins. In most cases, however, you would want to do this with the `post-receive` hook on the dedicated continuous integration server.

Another use case is to list information about all the `TODO` and `FIXME` comments in the code and documentation (for example, the author, version, file path, line number, and message), printing them to standard output of the hook, so that they are not forgotten and remain up to date and useful.

Hooks for applying patches from e-mails

You can set up three client-side hooks for the e-mail based workflow (where commits are sent in an e-mail). They are all invoked by the `git am` command (which name comes from the apply mailbox), which can be used to take saved e-mails with patches (created, for example, with `git format-patch` for example and sent, for example, with `git send-email`) and turn them into a series of commits. Those hooks are as follows:

1. The first hook to run is `applypatch-msg`. It is run after extracting the commit message from the patch and before applying the patch itself. As usual, for a hook which is not a `post-*` hook, Git aborts applying the patch if this hook exists with a nonzero status. It takes a single argument: the name of the temporary file with the extracted commit message.

You can use this hook to make sure that the commit message is properly formatted, or to normalize the commit message by having the script alter the file. The example `applypatch-msg` hook provided by Git simply runs the `commit-msg` hook if it exists as a hook (the file exists and is executable).

2. The next hook to run is `pre-applypatch`. It is run after the patch is applied to the working area, but before the commit is created. You can use it to inspect the state of the project before making a commit, for example, running tests. Exiting with a nonzero status aborts the

`git am` script without committing the patch.

The sample hook provided by Git simply runs the `pre-commit` hook, if present.

3. The last hook to run is `post-applypatch`, which runs after the commit is made. It can be used for notifying or logging, for example, notifying all the developers or just the author of the patch that you have applied it.

Other client-side hooks

There are a few other client-side hooks that do not fit into a series of steps in a single process.

The `pre-rebase` hook runs before you rebase anything. Like all the `pre-*` hooks, it can abort the rebase process with a nonzero exit code. You can use this hook to disallow rebasing (and thus rewriting) any commits that were already published. The hook is called with the name of the base branch (the upstream the series was forked from) and the name of the branch being rebased. The second parameter is passed to the hook only if the branch being rebased is not the current branch. The sample `pre-rebase` hook provided by Git tries to do this, though it makes some assumptions specific to Git's project development that may not match your workflow (take note that amending commits also rewrites them, and that rebasing may create a copy of a branch instead of rewriting it).

The `pre-push` hook runs during the `git push` operation, after it has checked the remote status (and exchange finding which revisions are absent on server), but before anything has been pushed. The hook is called with the reference to the remote (the URL or the remote name) and the actual push URL (the location of remote) as script parameters. Information about the commits to be pushed is provided on the standard input, one line per ref to be updated. You can use this hook to validate a set of ref updates before a push occurs; a nonzero exit code aborts the push. The example installed simply checks whether there are commits beginning with `WIP` in a set of revisions to be pushed or marked with the `nopush` keyword in the commit message, and when either of those is

true, it aborts the push. You can even make a hook prompt the user if he or she is sure. This hook complements the server-side checks, avoiding data transfer that would fail validation anyway.

The post-rewrite hook is run by commands that rewrite history (that replace commits), such as `git commit --amend` and `git rebase`. Note, however, that it is not run by large scale history rewriting, such as `git filter-branch`. The type of command that triggered the rewrite (`amend` or `rebase`) is passed as a single argument, while the list of rewrites is sent to the standard input. This hook has many of the same uses as the post-checkout and post-merge hooks, and it runs after automatic copying of notes, which is controlled by the `notes.rewriteRef` configuration variable (you can find more about notes mechanism in [Chapter 8, Keeping History Clean](#)).

The post-checkout hook is run after successful `git checkout` (or `git checkout <file>`) after having updated the worktree. The hook is given three parameters: the SHA-1s of the previous and current HEAD (which may or may not be different) and a flag indicating whether it was a whole project checkout (you were changing branches, the flag parameter is 1) or a file checkout (retrieving files from the index or named commit, the flag parameter is 0). As a special case, during initial checkout after `git clone`, this hook passes the all-zero SHA-1 as the first parameter (as a source revision). You can use this hook to set up your working directory properly for your use case. This may mean handling large binary files outside the repository (as an alternative to per-file the `filter` Git attribute) that you don't want to have in the repository, or setting the working directory metadata properties such as full permissions, owner, group, times, extended attributes, or ACLs. It can also be used to perform repository validity checks, or enhance the `git checkout` output by auto-displaying the differences (or just the diff statistics) from the previous checked out revision (if they were different).

The post-merge hook runs after a successful merge operation. You can use it in a way similar to post-checkout to restore data and metadata in the working tree that Git doesn't track, such as full permissions data (or

just make it invoke `post-checkout` directly). This hook can likewise validate the presence of files external to Git control that you might want copied in when the working tree changes.

For Git, objects in the repository (for example, commit objects representing revisions) are immutable; rewriting history (even amending a commit) is in fact creating a modified copy and switching to it, leaving the pre-rewrite history abandoned. Deleting a branch also leaves abandoned history. To prevent the repository from growing too much, Git occasionally performs garbage collection by removing old unreferenced objects. In all but ancient Git, this is done as a part of normal Git operations by them invoking `git gc --auto`. The `pre-auto-gc` hook is invoked just before garbage collection takes place and can be used to abort the operation, for example, if you are on battery power. It can also be used to notify you that garbage collection is happening.

Server-side hooks

In addition to the client-side hooks, which are run in your own repository, there are a couple of important server-side hooks that a system administrator can use to enforce nearly any kind of policy for your project.

These hooks are run before and after you do a push to the server. The pre hooks (as mentioned earlier) can exit nonzero to reject a push or part of it; messages printed by the pre hooks will be sent back to the client (sender). You can use these hooks to set up complex push policies. Git repository management tools, such as `gitolite` and Git hosting solutions, use these to implement more involved access control for repositories. The post hooks can be used for notification, starting a build process (or just to rebuild and redeploy the documentation) or running a full test suite, for example as a part of a continuous integration solution.

While writing server-side hooks, you need to take into account where in the sequence of operations does the hook take place and what information is available there, both as parameters or on the standard input, and in the repository.

That's what happens on the server when it receives a push:

1. Simplifying it a bit, the first step is that all the objects that were present in the client and missing on the server are sent to the server and stored (but are not yet referenced). If the receiving end fails to do this correctly (for example, because of the lack of disk space), the whole push operation will fail.
2. The `pre-receive` hook is run. It takes a list describing the references that are being pushed on its standard input. If it exits with a nonzero status, it aborts the whole operation and none of the references that were pushed are accepted.
3. For each ref being updated, the following happens:
 1. The built-in sanity checks may reject the push to the ref, including the check for an update of a checked out branch, or a non-fast-forward push (unless forced), and so on
 2. The `update` hook is run, passing ref to be pushed in arguments; if this script exits nonzero, only this ref will be rejected the sample hook blocks unannotated tags from entering the repository.
 3. The ref is updated (unless, in modern Git, the push is requested to be atomic)
4. If the push is atomic, all the refs are updated (if none were rejected).
5. The `post-receive` hook is run, taking the same data as the `pre-receive` one. This one can be used to update other services (for example, notify continuous integration servers) or notify users (via an e-mail or a mailing list, IRC, or a ticket-tracking system).
6. For each ref that was updated, the `post-update` hook is run. This can also be used for logging. The sample hook runs `git update-server-info` to prepare a repository, saving extra information to be used over *dumb* transports, though it would work better if run once as `post-receive`.
7. If push tries to update the currently checked out branch and the `receive.denyCurrentBranch` configuration variable is set to `updateInstead`, then `push-to-checkout` is run.

You need to remember that in pre hooks, you don't have refs updated

yet, and that post hooks cannot affect the result of an operation. You can use pre hooks for access control (permission checking), and post hooks for notification and updating side data and logs.

You will see example hooks (server-side and client-side) for the Git-enforced policy in [Chapter 11](#), *Git Administration*. You will also learn how other tools use those hooks, for example, for use in access control and triggering actions on push.

Extending Git

Git provides a few mechanisms to extend it. You can add shortcuts and create new commands, and add support for new transports; all without requiring to modify Git sources.

Command aliases for Git

There is one little tip that can make your Git command-line experience simpler, easier, and more familiar, namely, Git aliases. It is very easy in theory to create an alias. You simply need to create an `alias.<command-name>` configuration variable; its value is the expansion of alias.

One of the uses for aliases is defining short abbreviations for commonly used commands and their arguments. Another is creating new commands. Here are a couple of examples you might want to set up:

```
$ git config --global alias.co checkout
$ git config --global alias.ci commit
$ git config --global alias.lg log --graph --oneline --decorate
$ git config --global alias.aliases 'config --get-regexp
^alias\.'
```

The preceding setup means that typing, for example, `git ci` would be the same as typing `git commit`. Aliases take arguments just as the regular Git commands do. Git does not provide any default aliases that are defining shortcuts for the common operations, unless you use a friendly fork of Git by Felipe Contreras: `git-fc`.

Arguments are split by space, the usual shell quoting and escaping is supported; in particular, you can use a quote pair ("a b") or a backslash (a\ b) to include space in a single argument.

Note

Note, however, that you cannot have the alias with the same name as a Git command; in other words, you cannot use aliases to change the behavior of commands. The reasoning behind this restriction is that it

could make existing scripts and hooks fail unexpectedly. Aliases that hide existing Git commands (with the same name as Git commands) are simply ignored.

You might, however, want to run external command rather than a Git command in an alias. Or, you might want to join together the result of a few separate commands. In this case, you can start the alias definition with the ! character (with the exclamation mark):

```
$ git config --global alias.unmerged \
'!git ls-files --unmerged | cut -f2 | sort -u'
```

Because here the first command of the expansion of an alias can be an external tool, you need to specify the `git` wrapper explicitly, as shown in the preceding example.

Note

Note that in many shells, for example, in `bash`, ! is the history expansion character and it needs to be escaped as `\!`, or be within single quotes.

Note that such shell command will be executed from the top-level directory of a repository (after `cd` to a top level), which may not necessarily be the current directory. Git sets the `GIT_PREFIX` environment variable to the current directory path relative to the top directory of a repository, that is, `git rev-parse --show-prefix`. As usual, `git rev-parse` (and some `git` wrapper options) may be of use here.

The fact mentioned earlier can be used while creating aliases. The `git serve` alias, running `git daemon` to read-only serve the current repository at `git://127.0.0.1/`, makes use of the fact that the shell commands in aliases are executed from the top-level directory of a repo:

```
[alias]
serve = !git daemon --reuseaddr --verbose --base-path=. --
export-all ./git
```

Sometimes, you need to reorder arguments, use an argument twice, or

pass an argument to the command early in the pipeline. You would want to refer to subsequent arguments as \$1, \$2, and so on, or to all arguments as \$@, just like in shell scripts. One trick that you can find in older examples is to run a shell with a -c argument, like in the first of the examples mentioned next; the final dash is so that the arguments start with \$1, not with \$0. A more modern idiom is to define and immediately execute a shell function, like in the second example (it is a preferred solution because it has one of the fewer level of quoting, and lets you use standard shell argument processing):

```
[alias]
record-1 = !sh -c 'git add -p -- $@ && git commit' -
record-2 = !f() { git add -p -- $@ && git commit }; f
```

Aliases are integrated with command-line completion. While determining which completion to use for an alias, Git searches for a git command, skipping an opening brace or a single quote (thus, supporting both of the idioms mentioned earlier). With modern Git (version 2.1 or newer), you can use the null command ":" to declare the desired completion style. For example, alias expanding to !f() { : git commit ; ... } f would use a command completion for git commit, regardless of the rest of the alias.

Git aliases are also integrated with the help system; if you use the --help option on an alias, Git would tell you its expansion (so you can check the relevant man page):

```
$ git co --help
'git co' is aliased to `checkout'
```

Adding new Git commands

Aliases are best at taking small one-liners and converting them into small useful Git commands. You can write complex aliases, but when it comes to larger scripts, you would probably like to incorporate them into Git directly.

Git subcommands can be standalone executables that live in the Git execution path (which you can find by running git --exec-path); on

Linux, this normally is `/usr/libexec/git-core`. The `git` executable itself is a thin wrapper that knows where the subcommands live. If `git foo` is not a built-in command, the wrapper searches for the `git-foo` command first in the Git `exec` path, then in the rest of your `$PATH`. The latter makes it possible to write local Git extensions (local Git commands) without requiring access to the system's space.

This feature is what it makes possible to have user interface more or less integrated with the rest of Git in projects such as `git imerge` (see [Chapter 7, Merging Changes Together](#)) or `git annex` (see [Chapter 9, Managing Subprojects - Building a Living Framework](#)). It is also how projects such as Git Extras, providing extra Git commands, were made.

Note, however, that if you don't install the documentation for your command in typical places, or configure documentation system to find its help,, `git foo --help` won't work correctly.

Credential helpers and remote helpers

There is another place where simply dropping appropriately named executable enhances and extends Git. **Remote helper** programs are invoked by Git when it needs to interact with remote repositories and remote transport protocols Git does not support natively. You can find more about them in [Chapter 5, Collaborative Development with Git](#).

When Git encounters a URL of the form `<transport>://<address>`, where `<transport>` is a (pseudo)protocol that is not natively supported, it automatically invokes the `git remote-<transport>` command with a remote and full remote URL as arguments. A URL of the form `<transport>::<address>` also invokes this remote helper, but with just `<address>` as a second argument in the place of a URL. Additionally, with `remote.<remote-name>.vcs` set to `<transport>`, Git would explicitly invoke `git remote-<transport>` to access that remote.

The **helpers mechanism** in Git is about interacting with external scripts using a well-specified format.

Each remote helper is expected to support a subset of commands. You can find more information about the issue of creating new helpers in the `gitremote-helpers(1)` man page.

There is another type of helpers in Git, namely, **credentials helpers**. They can be used by Git to get the credentials from the user required, for example, to access the remote repository over HTTP. They are specified by the configuration though, just like the merge and diff drivers, and like the clean and smudge filters.

Summary

This chapter provided all the tools you need to use Git effectively. You got to know how to make the command-line interface easier to use and more effective with the Git-aware dynamic command prompt, command-line completion, autocorrection for Git commands, and using colors. You learned of the existence of alternative interfaces, from alternative porcelains to the various types of graphical clients.

You were reminded of the various ways to change the behavior of Git commands. You discovered how Git accesses its configuration, and learned about a selected subset of configuration variables. You have learned how to automate Git with hooks and how to make use of them. Finally, you have learned how to extend Git with new commands and support new URLs schemes.

This chapter was mainly about making Git more effective for you; the next chapter, [Chapter 11](#), *Git Administration*, would explain how to make Git more effective for other developers. You will find there more about server-side hooks and see their usage. You will also learn about repository maintenance.

Chapter 11. Git Administration

The previous chapter, *Customizing and Extending Git*, explained among others how to use Git hooks for automation moved earlier in the chapter. The client-side hooks were described in detail, while the server-side hooks were only sketched. Here, in this chapter, we will present the server-side hooks comprehensively, and mention the client-side hooks' usage as helpers.

The earlier chapters helped master your work with Git as a developer, as a person collaborating with others, and as a maintainer. When the book was talking about setting up repositories and branch structure, it was from the point of view of a Git user.

This chapter is intended to help readers who are in a situation of having to take up the administrative side of Git. This includes setting up remote Git repositories and configuring their access. It covers the work required to make Git go smoothly (that is, Git maintenance), and finding and recovering from the repository errors. This chapter will also describe transfer protocols and how to use server-side hooks for implementing and enforcing development policy. Additionally, you will find here a short description of the various types of tools that can be used to manage remote repositories, to help you choose among them.

In this chapter, we will cover the following topics:

- Server-side hooks—implementing a policy and notifications
- Transport protocols, authentication and authorization
- How to set up Git on the server
- Third-party tools for management of remote repositories
- Signed pushes to assert updating refs and enable audits
- Reducing the size of hosted repositories with alternates and namespaces
- Improving server performance and helping the initial clone
- Checking for repository corruption and fixing the repository
- Recovering from errors with the help of reflogs and `git fsck`

- Git repository maintenance and repacking
- Augmenting development workflows with Git

Repository maintenance

Occasionally, you may need to do some clean up of a repository, usually to make it more compact. Such clean ups are a very important step after migrating a repository from another version control system.

Modern Git (or rather all but ancient Git) from time to time runs the `git gc --auto` command in each repository. This command checks whether there are too many loose objects (objects stored as separate files, one file per object, rather than stored together in a packfile; objects are almost always created as loose), and if these conditions are met, then it would launch the garbage collection operation. The garbage collection means to gather up all the loose objects and place them in packfiles, and to consolidate many small packfiles into one large packfile. Additionally, it packs references into the `packed-refs` file. Objects that are unreachable even via reflog, and are safely old, do not get picked in the repack. Git would then delete loose objects and packfiles that got repacked (with some safety margin with respect to the age of the loose object's files), thus pruning old unreachable objects. There are various configuration knobs in the `gc.*` namespace to control garbage collection operations.

You can run `auto gc` manually with `git gc --auto`, or force garbage collection with `git gc`. The `git count-objects` command (perhaps with the help of the `-v` parameter) can be used to check whether there are signs that repack is needed. You can even run individual steps of the garbage collection individually with `git repack`, `git pack-refs`, `git prune`, and `git prune-packed`.

By default, Git would try to reuse the results of the earlier packing to reduce CPU time spent on the repacking, while still providing good disk space utilization. In some cases, you would want to more aggressively optimize the size of repository at the cost of it taking more time: this is possible with `git gc --aggressive` (or with repacking the repository by

hand with `git repack`, run with appropriate parameters). It is recommended to do this after import from other version control systems; the mechanism that Git uses for importing (fast-import stream) is optimized for the speed of the operation, not for the final repository size.

There are issues of maintenance not covered by `git gc`, because of their nature. One of them is pruning (deleting) remote-tracking branches that got deleted in the remote repository. This can be done with `git fetch --prune` or `git remote prune`, or on a per-branch basis with `git branch --delete --remotes <remote-tracking branch>`. This action is left to the user, and not run by `git gc`, because Git simply cannot know whether you have based your own work on the remote-tracking branch that is to be pruned.

Data recovery and troubleshooting

It is almost impossible to never make any mistakes. This applies also to using Git. The knowledge presented in this book, and your experience with using Git, should help in reducing the number of mistakes. Note that, Git tries quite hard not to make you lose your work; many mistakes are recoverable.

Recovering a lost commit

It may happen that you accidentally lost a commit. Perhaps, you force-deleted an incorrect branch that you were to be working on, or you rewound the branch to an incorrect place, or you were on an incorrect branch while starting an operation. Assuming something like this happened, is there any way to get your commits back and to undo the mistake?

Because Git does not delete objects immediately, but keeps them for a while, and only deletes them if they are unreachable during the garbage collection phase, the commit you lost will be there; you just need to find it. The garbage collection operation has, as mentioned, its own safeties; though if you find that you need troubleshooting, it would be better to turn off automatic garbage collection temporarily with `git config gc.auto never`.

Often, the simplest way to find and recover lost commits is to use the `git reflog` tool. For each branch, and separately for HEAD, Git silently records (logs) where the tip of the branch was in your local repository, at what time it was there, and how it got there. This record is called the reflog. Each time you commit or rewind a branch, the reflog for the branch and for the HEAD is updated. Each time you change the branches, the HEAD reflog is updated, and so on.

You can see where the tip of branch has been at any time by running `git`

reflog or git reflog <branch>. You can run git log -g instead, where -g is a short way of saying --walk-reflog; this gives you a normal configurable log output. There is also --grep-reflog=<pattern> to search the reflog:

```
$ git reflog
6c89dee HEAD@{0}: commit: Ping asynchronously
d996b71 HEAD@{1}: rebase -i (finish): returning to
refs/heads/ajax
d996b71 HEAD@{2}: rebase -i (continue): Ping asynchronously WIP
89579c9 HEAD@{3}: rebase -i (pick): Use Ajax mode
7c6d322 HEAD@{4}: commit (amend): Simplify index()
e1e6f65 HEAD@{5}: cherry-pick: fast-forward
eea7a7c HEAD@{6}: checkout: moving from ssh-check to ajax
c3e77bf HEAD@{7}: reset: moving to ajax@{1}
```

You should remember the <ref>@{<n>} syntax from [Chapter 2](#), *Exploring Project History*. With the information from reflogs, you can rewind the branch in question to the version from before the set of operations, or you can start a new branch starting with any commit in the list.

Let's assume that your loss was caused by deleting a wrong branch. Because of the way reflogs are implemented (because logs for a branch named `foo`, that is, for the `refs/heads/foo` ref, are kept in the `.git/logs/refs/heads/foo` file), reflog for a given branch is deleted together with the branch. You might still have the necessary information in the `HEAD` reflog, unless you have manipulated the branch tip without involving the working area, but it might not be easy to find it.

In the case when the information is not present in reflogs, one way to find the necessary information to recover lost objects is to use the `git fsck` utility, which checks your repository for integrity. With the `--full` option, you can use this command to show all unreferenced objects:

```
$ git fsck --full
Checking object directories: 100% (256/256), done.
Checking objects: 100% (58/58), done.
dangling commit 50b836cb93af955ca99f2ccd4a1cc4014dc01a58
dangling blob 59fc7435baf79180a3835dddc52752f6044bab99
dangling blob fd64375c1f2b17b735f3145446d267822ae3ddd5
```

[...]

You can see the SHA1 identifiers of the unreferenced (lost) commits in the lines with the `dangling commit` string prefix. To examine all these dangling commits, you can filter the `git fsck` output for the commits with `grep "commit"`, extract their SHA1 with `cut -d' ' -f3`, and then feed these revisions into `git log --stdin --no-walk`.

```
$ git fsck --full | grep "commit" | cut -d' ' -f3 | git log --stdin --no-walk
```

Troubleshooting Git

The main purpose of `git fsck` is to check for repository corruption. Besides having an option to find dangling objects, it runs sanity checks for each object and tracks the reachability fully. It can find corrupted and missing objects; if the corruption was limited to your clone and the correct version can be found in other repositories (in backups and other archives), you can try to recover those objects from an uncorrupted source.

Sometimes, however, the error might be deeper. You can try to find a Git expert outside your team, but often the data in the repository is proprietary. Creating a minimal reproduction of the problem is not always possible. With modern Git, if the problem is structural, you can try to use `git fast-export --anonymize` to strip the repository from the data while reproducing the issue.

If the repository is fine, but the problem is with the Git operations, you can try to use various tracking and debugging mechanisms built into Git, or you can try to increase the verbosity of the commands. You can turn on tracing with the appropriate environment variables, shown later. The trace output can be written to standard error stream by setting the value of the appropriate environment variable to `1`, `2`, or `true`. Other integer values between `2` and `10` will be interpreted as open file descriptors to be used for trace output. You can also set such environment variables to the absolute path of the file to write trace messages to.

These tracking-related variables include the following (see the manpage of the `git` wrapper for the complete list):

- `GIT_TRACE`: This enables general trace messages, which do not fit into any specific category. This includes the expansion of the Git aliases (see [Chapter 10, Customizing and Extending Git](#)), built-in command execution, and external command execution (such as pager, editor, or helper).
- `GIT_TRACE_PACKET`: This enables packet-level tracking of the network operations for the "smart" transport protocols. This can help with debugging protocol issues or the troubles with the remote server that you set up. For debugging and fetching from shallow repositories, there is `GIT_TRACE_SHALLOW`.
- `GIT_TRACE_SETUP`: This enables trace messages, printing information about the location of the administrative area of the repository, the working area, and the current working directory and the prefix (the last one is the subdirectory inside the repository directory structure).
- `GIT_TRACE_PERFORMANCE`: This shows the total execution time of each Git command.

There is also `GIT_CURL_VERBOSE` to emit all the messages generated by the `curl` library for the network operations over HTTP, and `GIT_MERGE_VERBOSITY` to control the amount of output shown by the recursive merge strategy.

Git on the server

The previous chapters should give you enough knowledge to master most of the day-to-day version control tasks. The [Chapter 5](#), *Collaborative Development with Git*, explained how one can lay out repositories for the collaboration. Here, we will explain how to actually set up remote Git repositories to serve.

The topic of administration of the Git repositories covers a large area. There are books written about specific repository management solutions, such as Gitolite, Gerrit, GitHub, or GitLab. Here, you will hopefully find enough information to help you with choosing a solution, or with crafting your own.

Let's start with the tools and mechanisms to manage remote repositories themselves, and then move on to the ways of serving Git repositories (putting Git on the server).

Server-side hooks

Hooks that are invoked on the server can be used for server administration; among others, these hooks can control the access to the remote repository by performing the authorization step, and can ensure that the commits entering the repository meet certain minimal criteria. The latter is best done with the additional help of client-side hooks, which were described in [Chapter 10](#), *Customizing and Extending Git*. That way users are not left with being notified that their commits do not pass muster only at the time they want to publish them. On the other hand, client-side hooks implementing validation are easy to skip with the `--no-verify` option (so server-side validation is necessary), and you need to remember to install them.

Note

Note, however, that server-side hooks are invoked only during push; you need other solutions for access control to fetch (and clone).

Hooks are also obviously not run while using *dumb* protocols—there is no Git on the server invoked then.

While writing hooks to implement some Git-enforced policy, you need to remember at what stage the hook in question is run and what information is available then. It is also important to know how the relevant information is passed to the hook—but you can find the last quite easily in the Git documentation in the `githooks` manpage. The previous chapter included a simple summary of server-side hooks. Here, we will expand a bit on this matter.

All the server-side hooks are invoked by `git receive-pack`, which is responsible for receiving published commits (which are received in the form of the packfile, hence the name of the command). For each hook, except for the `post-*` ones, if the hook exits with the nonzero status, then the operation is interrupted and no further stages are run. The post hooks are run after the operation finishes, so there is nothing to interrupt.

Both the standard output and the standard error output are forwarded to `git send-pack` at the client end, so the hooks can simply pass messages for the user by printing them (for example with `echo`, if the hook was written as a shell script). Note that the client doesn't disconnect until all the hooks complete their operation, so be careful if you try to do anything that may take a long time, such as automated tests. It is better to have a hook just start such long operations asynchronously and exit, allowing the client to finish.

You need to remember that, in pre hooks, you don't have refs updated yet, and that post hooks cannot affect the result of an operation. You can use pre hooks for access control (permission checking), and post hooks for notification, updating the side data, and logging. Hooks are listed in the order of operation.

The pre-receive hook

The first hook to run is `pre-receive`. It is invoked just before you start

updating refs (branches, tags, notes, and so on) in the remote repository, but after all the objects are received. It is invoked once for the receive operation. If the server fails to receive published objects, for example, because of the lack of the disk space or incorrect permissions, the whole `git push` operation will fail before Git invokes this hook.

This hook receives no arguments; all the information is received on the standard input of the script. For each ref to be updated, it receives a line in the following format:

```
<old-SHA1-value> <new-SHA1-value> <full-ref-name>
```

Refs to be created would have the old SHA1 value of 40 zeros, while refs to be deleted will have a new SHA1 value equal to the same. The same convention is used in all the other places, where the hooks receive the old and the new state of the updated ref.

This hook can be used to quickly bail out if the update is not to be accepted, for example, if the received commits do not follow the specified policy or if the signed push (more on this is mentioned later) is invalid. Note that to use it for access control, (for authorization) you need to get the authentication token somehow, be it with the `getpwuid` command or with an environment variable such as `USER`. This depends on the server setup and on the server configuration.

Push-to-update hook for pushing to nonbare repositories

When pushing to the nonbare repositories, if push tries to update the currently checked out branch then `push-to-checkout` will be run. This is done if the configuration variable `receive.denyCurrentBranch` is set to the `updateInstead` value (instead of one of the values: `true` or `refuse`, `warn` or `false`, or `ignore`) This hook receives the SHA1 identifier of the commit that is to be the tip of the current branch that is going to be updated.

This mechanism is intended to synchronize working directories when one side is not easily accessible interactively (for example, accessible via interactive `ssh`), or as a simple deploy scheme. It can be used to

deploy to a live website, or to run code tests on different operating systems.

If this hook is not present, Git will refuse the update of the ref if either the working tree or the index (the staging area) differs from `HEAD`, that is, if the status is "not clean". This hook is to be used to override this default behavior.

You can craft this hook to have it make changes to the working tree and to the index that are necessary to bring them to the desired state. For example, it can simply run `git read-tree -u -m HEAD "$1"` in order to switch to the new branch tip (the `-u` option updates the files in the worktree), while keeping the local changes (the `-m` option makes it perform a fast-forward merge with two commits/trees). If this hook exits with a nonzero status, then it will refuse pushing to the currently checked out branch.

The update hook

The next to run is the `update` hook, which is invoked *separately* for each ref being updated. This hook is invoked after the non-fast-forward check (unless the push is forced), and the per-ref built-in sanity checks that can be configured with `receive.denyDeletes`, `receive.denyDeleteCurrent`, `receive.denyCurrentBranch`, and `receive.denyNonFastForwards`.

Note that exiting with nonzero refuses the ref to be updated; if the push is atomic, then refusing any ref to be updated will abandon the whole push. With an ordinary push, only the update of a single ref will be refused; the push of other refs will proceed normally.

This hook receives the information about the ref to be updated as its parameters, in order: the full name of the ref being updated, the old SHA1 object name stored in the ref before the push, and the new SHA1 object name to be stored in the ref after the push.

The example `update.sample` hook can be used to block unannotated tags from entering the repository, and to allow or deny deleting and modifying tags, and deleting and creating branches. All the configurable

of this sample hook is done with the appropriate `hooks.*` configuration variables, rather than being hard-coded. There is also the `update-paranoid` Perl script in `contrib/hooks/`, which can be used as an example on how to use this hook for the access control. This hook is configured with an external configuration file, where, among others, you can set up access so that only commits and tags from specified authors are allowed, and authors additionally have correct access permissions.

Many repository management tools, for example Gitolite, set up and use this hook for their work. You need to read the tool documentation if you want for some reason to run your own `update` hook moved earlier together with the one provided by such a tool.

The post-receive hook

Then, after all the refs are updated, the `post-receive` hook is run. It takes the same data as the `pre-receive` one. Only now, all the refs point to the new SHA1s. It can happen that another user has modified the ref after it was updated, but before this hook was able to evaluate it. This hook can be used to update other services (for example, notify the continuous integration server), notify users (via an e-mail or a mailing list, an IRC channel, or a ticket-tracking system), or log the information about the push for audit (for example, about signed pushes). It supersedes and should be used in the place of the `post-update` hook.

There is no default `post-receive` hook, but you can find the simple `post-receive-email` script, and its replacement `git-multimail`, in the `contrib/hooks/` area. These two example hooks are actually developed separately from Git itself, but for convenience they are provided with the Git source. `git-multimail` sends one e-mail summarizing each changed ref, one e-mail for each new commit with the changes—threaded (as a reply) to the corresponding ref change e-mail, and one `announce` e-mail for each new annotated tag; each of these is separately configurable with respect to the e-mail address used and, to some extent, also with respect to the information included in the e-mails.

To provide an example of third-party tools, `irker` includes the script to be used as the Git's `post-receive` hook to send notifications about the new changes to the appropriate IRC channel using the `irker` daemon (set up separately).

The post-update hook (legacy mechanism)

Then the `post-update` hook is run. Each ref that was actually successfully updated passes its name as one of parameters; this hook takes a variable number of parameters. This is only a partial information; you don't know what the original (old) and updated (new) values of the updated refs were, and the current position of the ref is prone to race conditions (as explained before). Therefore, if you actually need the position of the refs, `post-receive` hook is a better solution.

The sample hook runs `git update-server-info` to prepare a repository for use over the dumb transports, by creating and saving some extra information. If the repository is to be published, or copied and published to be accessible via plain HTTP or other walker-based transport, you may consider enabling it. However, in modern Git, it is enough to simply set `receive.updateServerInfo` to `true`, so that hook is no longer necessary.

Using hooks to implement the Git-enforced policy

The only way to truly enforce policy is to implement it using server-side hooks, either `pre-receive` or `update`; if you want a per-ref decision, you need to use the latter. Client-side hooks can be used to help developers pay attention to the policy, but these can be disabled, or skipped, or not enabled.

Enforcing the policy with server-side hooks

One part of the development policy could be requiring that each commit message adheres to the specified template. For example, one may require for each nonmerge commit message to include the Digital

Certificate of Origin in the form of the `Signed-off-by:` line, or that each commit refers to the issue tracker ticket by including a string that looks like `ref: 2387`. The possibilities are endless.

To implement such a hook, you first need to turn the old and new values for a ref (that you got by either reading them line by line from the standard input in `pre-receive`, or as the `update` hook parameters) into a list of all the commits that are being pushed. You need to take care of the corner cases: deleting a ref (no commits pushed), creating a new ref, and a possibility of non-fast-forward pushes (where you need to use the merge base as the lower limit of the revision range, for example, with the `git merge-base` command), pushes to tags, pushes to notes, and other nonbranch pushes. The operation of turning a revision range into a list of commits can be done with the `git rev-list` command, which is a low-level equivalent (plumbing) of the user-facing `git log` command (porcelain); by default, this command prints out only the SHA1 values of the commits in the specified revision range, one per line, and no other information.

Then, for each revision, you need to grab the commit message and check whether it matches the template specified in the policy. You can use another plumbing command, called `git cat-file`, and then extract the commit message from this command output by skipping everything before the first blank line. This blank line separates commit metadata in the raw form from the commit body:

```
$ git cat-file commit a7b1a955
tree 171626fc3b628182703c3b3c5da6a8c65b187b52
parent 5d2584867fe4e94ab7d211a206bc0bc3804d37a9
author Alice Developer <alice@example.com> 1440011825 +0200
committer Alice Developer <alice@example.com> 1440011825 +0200
```

Added COPYRIGHT file

Alternatively, you can use `git show -s` or `git log -1`, which are both porcelain commands, instead of `git cat-file`. However, you would then need to specify the exact output format, for example, `git show -s --format=%B <SHA1>`.

When you have these commit messages, you can then use the regular expression match or another tool on each of the commit messages caught to check whether it matches the policy.

Another part of the policy may be the restrictions on how the branches are managed. For example, you may want to prevent the deletion of long-lived development stage branches (see [Chapter 6, Advanced Branching Techniques](#)), while allowing the deletion of topic branches. To distinguish between them, that is to find out whether the branch being deleted is a topic branch or not, you can either include a configurable list of branches to manage strictly , or you can assume that topic branches always use the `<user>/<topic>` naming convention. The latter solution can be enforced by requiring the newly created branches, which should be topic branches only, to match this naming convention.

Conceivably, you could make a policy that topic branches can be fast-forwarded only if they are not merged in, though implementing checks for this policy would be nontrivial.

Usually, you have only specific people with the permission to push to the official repository of the project (having holding so-called commit bit). With the server-side hooks, you can configure the repository so that it allows anyone to push, but only to the special `mob` branch; all the other push access is restricted.

You can also use server-side hooks to require that only annotated tags are allowed in the repository, that tags are signed with a public key that is present in the specified key server (and thus, can be verified by other developers), and that tags cannot be deleted or updated. If needed, you can restrict signed tags to those coming from the selected (and configured) set of users, for example enforcing a policy that only one of the maintainers can mark a project for a release (by creating an appropriately named tag, for example, `v0.9`).

Early notices about policy violations with client-side hooks

It would be not a good solution to have a strict enforcement of

development policies, and not provide users with a way to help watch and fulfill the said policies. Having one's work rejected during push can be frustrating; to fix the issue preventing one from publishing the commit, one would have to edit their history. See [Chapter 8, Keeping History Clean](#) for details on how to do it.

The answer to that problem is to provide some client-side hooks that users can install, to have Git notify them immediately when they are violating the policy, which would make their changes rejected by the server. The intent is to help correct any problem as fast as possible, usually before committing the changes. These client-side hooks must be distributed somehow, as hooks are not copied when cloning a repository. Various ways to distribute these hooks are described in [Chapter 10, Customizing and Extending Git](#).

If there are any limitations on the contents of the changes, perhaps that some file might be changed only by specified developers, the warning can be done with `pre-commit`. The `prepare-commit-msg` hook (and the `commit.template` configuration variable) can provide the developer with the customized template to be filled while working on a commit message. You can also make Git check the commit message, just before the commit would be recorded, with the `commit-msg` hook. This hook would find out and inform you whether you have correctly formatted the commit message and if this message includes all the information required by the policy. This hook can also be used instead of or in addition to `pre-commit` to check whether you are not modifying the files you are not allowed to.

The `pre-rebase` hook can be used to verify that you don't try to rewrite history in a manner that would lead to non-fast-forward push (with `receive.deny Non-FastForwards` on the server, forcing push won't work anyway).

As a last resort, there is a `pre-push` hook, which can check for correctness before trying to connect to the remote repository.

Signed pushes

[Chapter 5](#), *Collaborative Development with Git*, includes a description of various mechanisms that a developer can use to ensure integrity and authenticity of their work: signed tags, signed commits, and signed merges (merging signed tags). All these mechanisms assert that the objects (and the changes they contain) came from the signer.

But signed tags and commits do not assert that the developer wanted to have a particular revision at the tip of a particular branch. Authentication done by the hosting site cannot be easily audited later, and it requires you to trust the hosting site and its authentication mechanism. Modern Git (version 2.2 or newer) allows you to sign pushes for this purpose.

Signed pushes require the server to set up `receive.certNonceSeed` and the client to use `git push --signed`. Handling of signed pushes is done with the server-side hooks.

The signed push certificate sent by client is stored in the repository as a blob object and is verified using GPG. The `pre-receive` hook can then examine various `GIT_PUSH_CERT_*` environment variables (see the `git-receive-pack` manpage for the details) to decide whether to accept or deny given signed push.

Logging signed pushes for audit can be done with the `post-receive` hook. You can have it send an e-mail notification about the signed push or have it append information about the push to a log file. The push, certificate that is signed includes an identifier for the client's GPG key, the URL of the repository, and the information about the operations performed on the branches or tags in the same format as the `pre-receive` and `post-receive` input.

Serving Git repositories

In [Chapter 5](#), *Collaborative Development with Git*, we have examined four major protocols used by Git to connect with remote repositories: local, HTTP, SSH (Secure Shell), and Git (the native protocol). This was done from the point of view of a client connecting to the repository,

discussing what these protocols are and which one to use if the remote repository offers more than one.

This chapter will offer the administrator's side of view, explaining how to set up moved later, rephrased Git repositories to be served with these different transport protocols. Here we will also examine, for each protocol, how the authentication and authorization look like.

Local protocol

This is the most basic protocol, where a client uses the path to the repository or the `file://` URL to access remotes. You just need to have a shared filesystem, such as an NFS or CIFS mount, which contains Git repositories to serve. This is a nice option if you already have access to a networked filesystem, as you don't need to set up any server.

Access to repositories using a file-based transport protocol is controlled by the existing file permissions and network access permissions. You need read permissions to fetch and clone, and write permissions to push.

In a later case, if you want to enable push, you'd better set up a repository in such way that pushing does not screw up the permissions. This can be helped by creating a repository with the `--shared` option to `git init` (or to `git clone`). This option allows users belonging to the same group to push into the repository by using the *sticky group ID* to ensure that the repositories stay available to all the group members.

The disadvantage of this method is that shared access to a networked filesystem is generally more difficult to set up and reach safely from multiple remote locations, than a basic network access and setting up appropriate server. Mounting the remote disk over the Internet can be difficult and slow.

This protocol does not protect the repository against accidental damage. Every user has full access to the repository's internal files and there is nothing preventing from accidentally corrupting the repository.

SSH protocol

SSH (Secure Shell) is a common transport protocol (common especially for Linux users) for self-hosting Git repositories. SSH access to servers is often already set up in many cases as a way to safely log in to the remote machine; if not, it is generally quite easy to set up and use. SSH is an authenticated and encrypted network protocol.

On the other hand, you can't serve anonymous access to Git repositories over SSH. People must have at least limited access to your machine over SSH; this protocol does not allow anonymous read-only access to published repositories.

Generally, there are two ways to give access to Git repositories over SSH. The first is to have a separate account on the server for each client trying to access the repository (though such an account can be limited and does not need full shell access, you can in this case use `git-shell` as a login shell for Git-specific accounts). This can be used both with ordinary SSH access, where you provide the password, or with a public-key login. In a one-account-per-user case, the situation with respect to the access control is similar to the local protocol, namely, the access is controlled with the filesystem permissions.

A second method is to create a single shell account, which is often the `git` user, specifically to access Git repositories and to use public-key login to authenticate users. Each user who is to have an access to the repositories would then need to send his or her SSH public key to the administrator, who would then add this key to the list of authorized keys. The actual user is identified by the key he or she uses to connect to the server.

Another alternative is to have the SSH server authenticated from an LDAP server, or some other centralized authentication scheme (often, to implement single sign-ons). As long as the client can get (limited) shell access, any SSH authentication mechanism can be used.

Anonymous Git protocol

Next is the Git protocol. It is served by a special really simple TCP

daemon, which listens on a dedicated port (by default, port 9418). This is (or was) a common choice for fast anonymous unauthenticated read-only access to Git repositories.

The Git protocol server, `git daemon`, is relatively easy to set up. Basically, you need to run this command, usually in a daemonized manner. How to run the daemon (the server) depends on the operating system you use. It can be an `upstart` script, a `systemd` unit file, or a `sysvinit` script. A common solution is to use `inetd` or `xinetd`.

You can remap all the repository requests as relative to the given path (a project root for the Git repositories) with `--base-path=<directory>`. There is also support for virtual hosting; see the `git-daemon` documentation for the detail. By default, `git daemon` would export only the repositories that have the `git-daemon-export-ok` file inside `gitdir`, unless the `--export-all` option is used. Usually, you would also want to turn on `--reuseaddr`, to allow the server to restart without waiting for the connection to time out.

The downside of the Git protocol is the lack of authentication and the obscure port it runs on (that may require you to punch a hole in the firewall). The lack of authentication is because by default it is used only for read access, that is for fetching and cloning repositories. Generally, it is paired with either SSH (always authenticated, never anonymous) or HTTPS for pushing.

You can configure it to allow for push (by enabling the `receive-pack` service with the `--enable=<service>` command-line option or, on a per repository basis, by setting the `daemon.receivePack` configuration to `true`), but it is generally not recommended. The only information available to hooks for implementing access control is the client address, unless you require all the pushes to be signed. You can run external commands in an access hook, but this would not provide much more data about the client.

Note

One service you might consider enabling is `upload-archive`, which serves `git archive --remote`.

The lack of authentication means not only that the Git server does not know who accesses the repositories, but also that the client must trust the network to not spoof the address while accessing the server. This transport is not encrypted; everything goes in the plain.

Smart HTTP(S) protocol

Setting up the so-called "smart" HTTP(S) protocol consists basically of enabling a server script that would invoke `git receive-pack` and `git upload-pack` on the server. Git provides a CGI script named `git-http-backend` for this task. This CGI script can detect if the client understands smart HTTP protocol; if not, it will fall back on the "dumb" behavior (a backward compatibility feature).

To use this protocol, you need some CGI server, for example, Apache (with this server you would also need the `mod_cgi` module or its equivalent, and the `mod_env` and `mod_alias` modules). The parameters are passed using environment variables (hence, the need for `mod_env` in case of using Apache): `GIT_PROJECT_ROOT` to specify where repositories are and an optional `GIT_HTTP_EXPORT_ALL` if you want to have all the repositories exported, not only those with the `git-daemon-export-ok` file in them.

The authentication is done by the web server. In particular, you can set it up to allow unauthenticated anonymous read-only access, while requiring authentication for push. Utilizing HTTPS gives encryption and server authentication, like for the SSH protocol. The URL for fetching and pushing is the same when using HTTP(S); you can also configure it so that the web interface for browsing Git repositories uses the same URL as for fetching.

Note

The documentation of `git-http-backend` includes a set up for Apache for different situations, including unauthenticated read and

authenticated write. It is a bit involved, because initial ref advertisements use the query string, while the `receive-pack` service invocation uses path info.

On the other hand, requiring authentication with any valid account for reads and writes and leaving the restriction of writes to the server-side hook is simpler and often acceptable solution.

If you try to push to the repository that requires authentication, the server can prompt for credentials. Because the HTTP protocol is stateless and involves more than one connection sometimes, it is useful to utilize credential helpers (see [Chapter 10, Customizing and Extending Git](#)) to avoid either having to give the password more than once for a single operation, or having to save the password somewhere on the disk (perhaps, in the remote URL).

Dumb protocols

If you cannot run Git on the server, you can still use the dumb protocol, which does not require it. The dumb HTTP(S) protocol expects the Git repository to be served as normal static files from the web server.

However, to be able to use this kind of protocol, Git requires the extra `objects/info/packs` and `info/refs` files to be present on the server, and kept up to date with `git update-server-info`. This command is usually run on push via one of the earlier mentioned smart protocols (the default `post-update` hook does that, and so does `git-receive-pack` if `receive.updateServerInfo` is set to true).

It is possible to push with the dumb protocol, but this requires a set up that allows updating files using a specified transport; for the dumb HTTP(S) transport protocol, this means configuring WebDAV.

Authentication in this case is done by the web server for static files. Obviously, for this kind of transport, Git's server-side hooks are not invoked, and thus they cannot be used to further restrict access.

Note

Note that for modern Git, the dumb transport is implemented using the `curl` family of remote helpers, which may be not installed by default.

This transport works (for fetch) by downloading requested refs (as plain files), examining where to find files containing the referenced commit objects (hence, the need for server information files, at least for objects in packfiles), getting them, and then walking down the chain of revisions, examining each object needed, and downloading new files if the object is not present yet in the local repository. This walker method can be horrendously inefficient if the repository is not packed well with respect to the requested revision range. It requires a large number of connections and always downloads the whole pack, even if only one object from it is needed.

With smart protocols, Git on the client-side and Git on the server negotiate between themselves which objects are needed to be sent (want/have negotiation). Git then creates a customized packfile, utilizing the knowledge of what objects are already present on the other side, and usually including only deltas, that is, the difference from what the other side has (a thin packfile). The other side rewrites the received packfile to be self-contained.

Remote helpers

Git allows us to create support for new transport protocols by writing remote helper programs. This mechanism can be also used to support foreign repositories. Git interacts with a repository requiring a remote helper by spawning the helper as an independent child process, and communicating with the said process through its standard input and output with a set of commands.

You can find third-party remote helpers to add support to the new ways of accessing repositories, for example, there is `git-remote-dropbox` to use Dropbox to store the remote Git repository. Note, however, that remote helpers are (possibly yet) limited in features as compared to the built-in transport support.

Tools to manage Git repositories

Nowadays, there is no need to write the Git repositories management solution yourself. There is a wide range of various third-party solutions that you can use. It is impossible to list them all, and even giving recommendations is risky. The Git ecosystem is actively developed; which tool is the best can change since the time of writing this.

I'd like to focus here just on the types of tools for administrators, just like it was done for GUIs in [Chapter 10, Customizing and Extending Git](#).

First, there are **Git repository management** solutions (we have seen one example of such in the form of the `update-paranoid` script in the `contrib/` area). These tools focus on access control, usually the authorization part, making it easy to add repositories and manage their permissions. An example of such a tool is Gitolite. They often support some mechanism to add your own additional access constraints.

Then, there are **web interfaces**, allowing us to view Git repositories using a web browser. Some make it even possible to create new revisions using a web interface. They differ in capabilities, but usually offer at least a list of available Git repositories, a summary view for each repository, an equivalent of the `git log` and `git show` commands, and a view with a list of files in the repository. An example of such tools is `gitweb` script in Perl that is distributed with Git; another is `cgit` used by <https://www.kernel.org/> for the Linux kernel repositories (and others).

Often useful are the **code review (code collaboration)** tools. These make it possible for developers in a team to review each other's proposed changes using a web interface. These tools often allow the creation of new projects and the handling of access management. An example of such a tool is Gerrit Code Review.

Finally, there are **Git hosting** solutions, usually with a web interface for the administrative side of managing repositories, allowing us to add

users, create repositories, manage their access, and often work from the web browser on the Git repositories. An example of such a tool is GitLab. There are also similar **source code management** systems, which provide (among other web-based interfaces) repository hosting services together with the features to collaborate and manage development. Here, Phabricator and Kallithea can be used as examples.

Of course, you don't need to self-host your code. There is a plethora of third-party hosted options: GitHub, Bitbucket, and so on. There are even hosted solutions using open source hosting management tools, such as GitLab.

Tips and tricks for hosting repositories

If you want to self-host Git repositories, there are a few things that may help you with server performance and user satisfaction.

Reducing the size taken by repositories

If you are hosting many forks (clones) of the same repository, you might want to reduce disk usage by somehow sharing common objects. One solution is to use alternates (for example, with `git clone --reference`) while creating a fork. In this case, the derived repository would look to its parent object storage if the object is not found on its own.

There are, however, two problems with this approach. First is that you need to ensure that the object the borrowing repository relies on does not vanish from the repository set as the alternate object storage (the repository you borrow from). This can be done, for example, by linking the borrowing repository refs in the repository lending the objects, for example, in the `refs/borrowed/` namespace. Second is that the objects entering the borrowing repository are not automatically de-duplicated: you need to run `git repack -a -d -l`, which internally passes the `--local` option to `git pack-objects`.

An alternate solution would be to keep every fork together in a single repository, and use `git namespaces` to manage separate views into the

DAG of revisions, one for each fork. With plain Git, this solution means that the repository is addressed by the URL of the common object storage and the namespace to select a particular fork. Usually, this is managed by a server configuration or by a repository management tool; such mechanism translates the address of the repository into a common repository and the namespace. The `git-http-backend` manpage includes an example configuration to serve multiple repositories from different namespaces in a single repository. Gitolite also has some support for namespaces in the form of logical and backing repositories and `option namespace.pattern`, though not every feature works for logical repositories.

Storing multiple repositories as the namespace of a single repository avoids storing duplicated copies of the same objects. It automatically prevents duplication between new objects without the need for ongoing maintenance, as opposed to the alternates solution. On the other hand, the security is weaker; you need to treat anyone with access to the single namespace, which is within the repository as if he or she had an access to all the other namespaces, though this might not be a problem for your case.

Speeding up smart protocols with pack bitmaps

Another issue that you can stumble upon while self-hosting repositories is the performance of smart protocols. For the clients of your server, it is important that the operations finish quickly; as an administrator, you would not want to generate high CPU load on the server due to serving Git repositories.

One feature, ported from JGit, should significantly improve the performance of the counting objects phase, while serving objects from a repository that uses this trick. This feature is a bitmap-index file, available since Git 2.0.

This file is stored alongside the packfile and its index. It can be generated manually by running `git repack -A -d --write-bitmap-index`, or be generated automatically together with the packfile by

setting the `repack.writeBitmaps` configuration variable to `true`. The disadvantage of this solution is that bitmaps take additional disk space, and the initial repack requires extra time to create bitmap-index.

Solving the large nonresumable initial clone problem

Repositories with a large codebase and a long history can get quite large. The problem is that the initial clone, where you need to get all of a possibly large repository, is an all-or-nothing operation, at least for modern (safe and effective) smart transfer protocols: SSH, `git://`, and smart HTTP(S). This might be a problem if a network connection is not very reliable. There is no support for a resumable clone, and it unfortunately looks like it is fundamentally hard problem to solve for Git developers. This does not mean, however, that you, as a hosting administrator, can do nothing to help users get this initial clone.

One solution is to create, with the `git bundle` command, a static file that can be used for the initial clone, or as reference repository for the initial clone (the latter can be done with the `git clone --reference= <bundle> --dissociate` command if you have Git 2.3 or a newer looks unnecessary). This bundle file can be distributed using any transport; in particular, one that can be resumed if interrupted, be it HTTP, FTP, rsync, or BitTorrent. The conventions people use, besides explaining how to get such a bundle in the developer documentation, is to use the same URL as for the repository, but with the `.bundle` extension (instead of an empty extension or a `.git` suffix).

There are also more esoteric approaches like a step by step deepening of a shallow clone (or perhaps, just using a shallow clone with `git clone --depth` is all that's needed), or using approaches such as GitTorrent.

Augmenting development workflows

Handling version control is only a part of the development workflow. There is also work management, code review and audit, running automated tests, and generating builds.

Many of these steps can be helped using specialized tools. Many of them offer Git integration. For example, code review can be managed using Gerrit, requiring that each change passes a review before being made public. Another example is setting up development environments so that pushing changes to the public repository can automatically close tickets in the issue tracker based on the patterns in the commit messages. This can be done with the server-side hooks or with the hosting service's webhooks.

A repository can serve as a gateway, running automated tests (for example, with the help of Jenkins/Hudson continuous integration service), and deploying changes to ensure quality environments only after passing all of these tests. Another repository can be configured to trigger builds for various supported systems. Many tools and services support push to deploy mechanisms (for example, Heroku or Google's App Engine).

Git can automatically notify users and developers about the published changes. It can be done via e-mail, using the mailing list or the IRC channel, or a web-based dashboard application. The possibilities are many; you only need to find them.

Summary

This chapter covered various issues related to the administrative side of working with Git. You have learned the basics of maintenance, data recovery, and repository troubleshooting. You have also learned how to set up Git on the server, how to use server-side hooks, and how to manage remote repositories. The chapter covered tips and tricks for a better remote performance. The information in this chapter should help you choose the Git repository management solution, or even write your own.

The next chapter will include a set of recommendations and best practices, both specific to Git and those that are version control agnostic. A policy based on these suggestions can be enforced and encouraged with the help of the tools described here in this chapter.

Chapter 12. Git Best Practices

The last chapter of *Mastering Git* presents a collection of generic and Git-specific version control recommendations and best practices. You have encountered many of these recommendations already in the earlier chapters; they are here as a summary and as a reminder. For details and the reasoning behind each best practice, you would be referred to specific chapters.

This chapter will cover issues of managing the working directory, creating commits and series of commits (pull requests), submitting changes for inclusion, and the peer review of changes.

In this chapter, we will cover the following topics:

- How to separate projects into repositories
- What types of data to store in a repository and which files should Git ignore
- What to check before creating a new commit
- How to create a good commit and a good commit series (or, in other words, how to create a good pull request)
- How to choose an effective branching strategy, and how to name branches and tags
- How to review changes and how to respond to the review

Starting a project

When starting a project, you should choose and clearly define a project governance model (who manages work, who integrates changes, and who is responsible for what). You should decide about the license and the copyright of the code: whether it is work for hire, whether contributions would require a copyright assignment, a contributor agreement, or a contributor license agreement, or simply a digital certificate of origin.

Dividing work into repositories

In centralized version control systems, often everything under the sun is put under the same project tree. With distributed version control systems such as Git, it is better to split separate projects into separate repositories.

There should be one conceptual group per repository; divide it beforehand correctly. If some part of the code is needed by multiple separate projects, consider extracting it into its own project and then incorporating it as a submodule or subtree, grouping concepts into a superproject. See [Chapter 9, Managing Subprojects - Building a Living Framework](#) for the details.

Selecting the collaboration workflow

You need to decide about the collaboration structure, whether your project would use a dispersed contributor model, a "blessed" repository model, or a central repository, and so on (as found in [Chapter 5, Collaborative Development with Git](#)). This often requires setting up an access control mechanism and deciding on the permission structure; see [Chapter 11, Git Administration](#) on how one can set up this.

You would also need to decide how to structure your branches; see [Chapter 6, Advanced Branching Techniques](#), for possible solutions. This decision doesn't need to be cast in stone; as your project and your team experience grows, you might want to consider changing the branching model, for example, from the plain branch-per-feature model to full Gitflow, or to GitHub-flow, or any of the other derivatives.

The decision about licensing, the collaboration structure, and the branching model should all be stated explicitly in the developer documentation (at minimum, including the `README` and `LICENSE/COPYRIGHT` files). You need to remember that if the way in which the project is developed changes, which can happen, for example, because the project has grown beyond its initial stage, this documentation would need to be kept up to date.

Choosing which files to keep under version

control

In most cases, you should not include any of the *generated files* in the version control system (though there are some very rare exceptions). Track only the sources (the original resources); Git works best if these sources are plain text files, but it works well also with binary files.

To avoid accidentally including unwanted files in a repository, you should use the **gitignore patterns**. These ignore patterns that are specific to a project (for example, results and by-products of a build system) should go into the `.gitignore` file in the project tree; those specific to the developer (for example, backup files created by the editor one uses or the operating system-specific helper files) should go into his or her per-user `core.excludesFile` (which, in modern Git, is the `~/.config/git/ignore` file), or into a local configuration of the specific clone of the repository, that is, `.git/info/excludes`. See [Chapter 4, Managing Your Worktree](#) for details.

A good start for ignore patters is the <https://www.gitignore.io> trailing slash is not necessary; choose whichever looks better. Website with the `.gitignore` templates for various operating systems, IDEs, and programming languages.

Another important rule is to not add to be tracked by Git the configuration files that might change from environment to environment (for example, being different for MS Windows and for Linux).

Working on a project

Here are some guidelines on how to create changes and develop new revisions. These guidelines can be used either for your own work on your own project, or to help contribute your code to the project maintained by somebody else.

Different projects can use different development workflows; therefore, some of the recommendations presented here might not make sense, depending on the workflow that is used for a given project.

Working on a topic branch

Branching in Git has two functions ([Chapter 6, Advanced Branching Techniques](#)): as a mediator for the code contributed by developers keeping to the specified level of code stability and maturity (long-running public branches), and as a sandbox for the development of a new idea (short-lived private branches).

The ability to sandbox changes is why it is considered a good practice to create a separate branch for each new task you work on. Such a branch is called a **topic branch** or a **feature branch**. Using separate branches makes it possible to switch between tasks easily, and to keep disparate work in progress from interfering with each other.

You should choose short and descriptive names for branches. There are different naming conventions for topic branches; the convention your project uses should be specified in the developer documentation. In general, branches are usually named after a summary of a topic they host, usually all lower-case and with spaces between words replaced by hyphens or underscores (see the `git-check-ref-format(1)` manpage to know what is forbidden in branch names). Branch names can include slash (be hierarchical).

If you are using an issue tracker, then the branch which fixes a bug, or implements an issue, can have its name prefixed with the identifier (the

number) of the ticket describing the issue, for example, 1234-doc_spellcheck. On the other hand, the maintainer, while gathering submissions from other developers, could put these submissions in topic branches named after the initials of the developer and the name of the topic, for example, ad/whitespace-cleanup (this is an example of **hierarchical branch name**).

It is considered a good practice to delete your branch from your local repository, and also from the upstream repository after you are done with the branch in question to reduce clutter.

Deciding what to base your work on

As a developer, you would be usually working at a given time on some specific topic, be it a bug fix, enhancement or correction to some topic, or a new feature.

Decision about where to start your work on a given topic, and what branch to base your work on, depends on the branching workflow chosen for a project (see [Chapter 6, Advanced Branching Techniques](#) for a selection of branchy workflows). This decision also depends on the type of the work you do.

For a topic branch workflow (or a branch-per-feature workflow), you would want to base your work on the oldest and most stable long-running branch that your change is relevant to, and for which you plan to have your changes merged into. This is because, as described in [Chapter 6, Advanced Branching Techniques](#), you should never merge a less stable branch into a more stable branch. The reason behind this best practice rule is to not destabilize branch, as merges bring all the changes.

Different types of changes require a different long-lived branch to be used as a base for a topic branch with those changes, or to put those changes onto. In general, to help developers working on a project, this information should be described in the developer documentation; not everybody needs to be knowledgeable about the branching workflow used by the project.

The following describes what is usually used as a base branch, depending on purpose of changes:

- **Bugfix:** In this case the topic branch (the bugfix branch) should be based on the oldest and the most stable branch in which the bug is present. This means, in general, starting with the maintenance branch. If the bug is not present in the maintenance branch, then base the bugfix branch on the stable branch. For a bug that is not present in the stable branch, find the topic branch that introduced it and base your work on top of that topic branch.
- **New feature:** In this case the topic branch (the feature branch) should be based on the stable branch, if possible. If the new feature depends on some topic that is not ready for the stable branch, then base your work on that topic (from a topic branch).
- **Corrections and enhancements:** To a topic that didn't get merged into the stable branch should be based on the tip of the topic branch being corrected. If the topic in question is not considered published, it's all right to make changes to the steps of the topic, squashing minor corrections in the series (see the section about rewriting history in [Chapter 8, Keeping History Clean](#)).

If the project you are contributing to is large enough to have dedicated maintainers for selected parts (subsystems) of the system, you first need to decide which repository and which fork (sometimes named "a tree") to base your work on.

Splitting changes into logically separate steps

Unless your work is really simple and it can be done in a single step (a single commit)—like many of the bug fixes—you should make separate commits for the logically separate changes, one commit per single step. Those commits should be ordered logically.

Following a good practice for a commit message (with an explanation of what you have done—see the next section) could help in deciding when to commit. If your description gets too long and you begin to see that you have two independent changes squished together, that's a sign that

you probably need to split your commit to finer grained pieces and use smaller steps.

Remember, however, that it is a matter of balance, of the project conventions, and of the development workflow chosen. Changes should, at minimum, stand on their own. At each step (at each commit) of the implementation of a feature, the code compiles and the program passes the test suite. You should commit early and often. Smaller self-contained revisions are easier to review, and with smaller but complete changes, it is easier to find regression bugs with `git bisect` (which is described in [Chapter 2, Exploring Project History](#)).

Note that you don't necessarily need to come up with the perfect sequence of steps from the start. In the case when you notice that you have entangled the work directory's state, you can make use of the staging area, using an interactive add to disentangle it (this is described in [Chapter 3, Developing with Git](#) and [Chapter 4, Managing Your Worktree](#)). You can also use an interactive rebase or similar techniques, as shown in [Chapter 8, Keeping History Clean](#), to curate commits into an easy-to-read (and easy-to-bisect) history before publishing.

You should remember that a commit is a place to record your result (or a particular step towards the result), not a place to save the temporary state of your work. If you need to temporarily save the current state before going back to it, use `git stash`.

Writing a good commit message

A good commit message should include an explanation for change that is detailed enough, so that other developers on the team (including reviewers and the maintainer) can judge if it is a good idea to include the change in the codebase. This *good or not* decision should not require them to read actual changes to find out what the commit intends to do.

The first line of the commit message should be a short, terse description (around from 50 to 72 characters) with the summary of changes. It should be separated by an empty line from the rest of the commit

message, if there is one. This is partially because, in many places, such as in the `git log --oneline` command output, in the graphical history viewer like `gitk`, or in the instruction sheet of `git rebase --interactive`, you would see only this one line of the commit message and you would have to decide about the action with respect to that commit on the basis of this one line. If you have trouble with coming up with a good summary of changes, this might mean that these changes need to be split into smaller steps.

There are various conventions for this summary line of changes. One convention is to prefix the first summary line with `area:`, where area is an identifier for the general area of the code being modified: a name of the subsystem, of an affected subdirectory, or a filename of a file being changed. If the development is managed via an issue tracker, this summary line can start with something like the `[#1234]` prefix, where `1234` is the identifier of an issue or a task implemented in the commit. In general, when not sure about what information to include in the commit message, refer to the development documentation, or fall back to the current convention used by other commits in the history.

Note

If you are using Agile development methods, you can look for especially good commit messages during retrospectives, and add them as examples in the developer documentation for the future.

For all but trivial changes, there should be a longer meaningful description, the body of the commit message. There is something that people coming from other version control systems might need to unlearn: namely, not writing a commit message at all or writing it all in one long line. Note that Git would not allow to create a commit with an empty commit message unless forced to with `--allow-empty`.

The commit message should:

- Include the rationale for the commit, explain the problem that the commit tries to solve, the **why**: in other words, it should include description of what is wrong with the current code or the current

behavior of the project without the change; this should be self-contained, but it can refer to other sources like the issue tracker (the bug tracker), or other external documents such as articles, wikis, or Common Vulnerabilities and Exposures (CVE).

- Include a quick summary. In most cases, it should also explain (the **how**) and justify the way the commit solves the problem. In other words, it should describe why do you think the result with the change is better; this part of the description does not need to explain what the code does, that is largely a task for the code comments.
- If there was more than one possible solution, include a description of the alternate solutions that were considered but were ultimately discarded, perhaps with links to the discussion or review(s).

It's a good idea to try to make sure that your explanation for changes can be understood without access to any external resources (that is, without an access to the issue tracker, to the Internet, or to the mailing list archive). Instead of just referring to the discussion, or in addition to giving a URL or an issue number, write a summary of the relevant points in the commit message.

One of the possible recommendations to write the commit message is to describe changes in the imperative mood, for example, `make foo do bar`, as if you are giving orders to the codebase to change its behavior, instead of writing `This commit makes ... or [I] changed`

Here, `commit.template` and commit message hooks can help in following these practices. See [Chapter 10, Customizing and Extending Git](#), for details (and [Chapter 11, Git Administration](#), for a description of the way to enforce this recommendation).

Preparing changes for submission

Consider rebasing the branch to be submitted on top of the current tip of the base branch. This should make it easier to integrate changes in the future. If your topic branch was based on the development version, or on the other in-flight topic branch (perhaps because it depended on some specific feature), and the branch it was based on got merged into a

stable line of development, you should rebase your changes on top of the stable integration branch instead.

The time of rebase is also a chance for a final clean-up of the history; the chance to make submitted changes easier to review. Simply run an interactive rebase, or a patch management tool if you prefer it (see [Chapter 8, Keeping History Clean](#)). One caveat: *do not rewrite the published history*.

Consider testing that your changes merge cleanly, and fix it if they don't, if the fix is possible. Make sure that they would cleanly apply, or cleanly merge into the appropriate integration branch.

Take a last look at your commits to be submitted. Make sure that your changes do not add the commented out (or the `ifdef-ed out`) code, and it does not include any extra files not related to the purpose of the patch (for example, that they do not include the changes from the next new feature). Review your commit series before submission to ensure accuracy.

Integrating changes

The exact details on how to submit changes for merging depends, of course, on the development workflow that the project is using. Various classes of possible workflows are described in [Chapter 5, Collaborative Development with Git](#).

Submitting and describing changes

If the project has a dedicated maintainer or, at least, if it has someone responsible to merge the proposed changes into the official version, you would need also to describe submitted changes as a whole (in addition to describing each commit in the series). This can be done in the form of a cover letter for the patch series while sending changes as patches via e-mail; or it can be comments in the pull request while using collocated contributor repositories model; or it can be the description in an e-mail with a pull request, which already includes the URL and the branch in your public repository with changes (generated with `git request-pull`).

This cover letter or a pull request should include the description of the purpose of the patch series or the pull request. Consider providing there an overview of why the work is taking place (with any relevant links and a summary of the discussion). Be explicit with stating that it is a work in progress, saying this in the description of changes.

In the dispersed contributor model, where changes are submitted for review as patches or patch series, usually to the mailing list, you should use Git-based tools such as `git format-patch` and, if possible, `git send-email`. Multiple related patches should be grouped together, for example, in their own e-mail thread. The convention is to send them as replies to an additional cover letter message, which should describe the feature as a whole.

If the changes are sent to the mailing list, it is a common convention to prefix your subject line with `[PATCH]` or with `[PATCH m/n]` (where `m` is the patch number in the series of the `n` patches). This lets people easily

distinguish patch submissions from other e-mails. This part can be done with `git format-patch`. What you need to decide yourself is to whether to use additional markers after `PATCH` to mark the nature of the series, for example, `PATCH/RFC` (`RFC` means here Request for Comments, that is an idea for a feature with an example of its implementation; such patch series should be examined if the idea is worthy; it is not ready to be applied/merged but provided only for the discussion among developers).

In the collocated contributor repositories model, where all the developers use the same Git hosting website or software (for example, GitHub, Bitbucket, GitLab, or a private instance of it, and so on), you would push changes to your own public repository, a *fork* of the official version. Then, you would create a *merge request* or a *pull request*, usually via a web interface of the hosting service, again describing the changes as a whole there.

In the case of using the central repository (perhaps, in a shared maintenance model), you would push changes to a separate and possibly new branch in the integration repository, and then send an announcement to the maintainer so that he or she would be able to find where the changes to merge are. The details of this step depends on the exact setup; sending announcement might be done via e-mail, via some kind of internal messaging mechanism, or even via tickets (or via the comments in the tickets).

The development documentation might include rules specifying to where and to what place to send announcements and/or changes to. It is considered a courtesy to notify the people who are involved in the area of code you are touching about the new changes (here you can use `git blame` and `git shortlog` to identify these people; see [Chapter 2, Exploring Project History](#)). These people are important; they can write a comment about the change and help reviewing it.

Tip

Crediting people and signing your work

Some open source projects, in order to improve the tracking provenance of the code, use the sign-off procedure borrowed from the Linux kernel called Digital Certificate of Origin. The sign-off is a simple line at the end of the commit message, saying for example:

Signed-off-by: Random Developer <rdeveloper@company.com>

By adding this line, you certify that the contribution is either created as a whole or in part by you, or is based on the previous work, or was provided directly to you, and that everybody in the chain have the right to submit it under appropriate license. If your work is based on the work by somebody else, or if you are just passing somebody's work, then there can be multiple sign-off lines, forming a chain of provenance.

In order to credit people who helped with creating the commit, you can append to the commit message other trailers, such as `Reported-by:`, `Reviewed-by:`, `Acked-by:` (this one states that it was liked by the person responsible for the area covered by the change), or `Tested-by:`.

The art of the change review

Completing a peer review of changes is time-consuming (but so is using version control), but the benefits are huge: better code quality, reducing the time needed for quality assurance testing, transfer of knowledge, and so on. The change can be reviewed by a peer developer, or reviewed by a community (requiring consensus), or reviewed by the maintainer or one of his/her lieutenants.

Before beginning the code review process, you should read through the description of the proposed changes to discover why the change was proposed, and decide whether you are the correct person to perform the review (that is one of reasons why good commit messages are so important). You need to understand the problem that the change tries to solve. You should familiarize yourself with the context of the issue, and with the code in the area of changes.

The first step is to reproduce the state before the change and check whether the program works as described (for example, that the bug in a

bugfix can be reproduced). Then, you need to check out the topic branch with proposed changes and verify that the result works correctly. If it works, review the proposed changes, creating a comprehensive list of everything wrong (though if there are errors early in the process, it might be unnecessary to go deeper), as follows:

- Are the commit messages descriptive enough? Is the code easily understood?
- Is the contribution architected correctly? Is it architecturally sound?
- Does the code comply with project's coding standards and with the agreed upon coding conventions?
- Are the changes limited to the scope described in the commit message?
- Does the code follow the industry's best practices? Is it safe and efficient?
- Is there any redundant or duplicate code? Is the code as modular as possible?
- Does the code introduce any regressions in the test suite? If it is a new feature, does the change include the tests for the new feature, both positive and negative?
- Is the new code as performing the way it did before the change (within the project's tolerances)?
- Are all the words spelled correctly, and does the new version follow the formatting guidelines for the content?

This is only one possible proposal for such code review checklist. Depending on the specifics of the project, there might be more questions that need to be asked as a part of the review; make the team write your own checklist. You can find good examples online, such as Fog Creek's Code Review Checklist.

Divide the problems that you have found during reviews into the following categories:

- **Wrong problem:** This feature does not lie within the scope of project. It is used sometimes for the bug that cannot be reproduced. Is the idea behind the contribution sound? If so, eject changes with or without prejudice, do not continue the analysis for the review.

- **Does not work:** This does not compile, introduces a regression, doesn't pass the test suite, doesn't fix the bug, and so on. These problems absolutely must be fixed.
- **Fails best practices:** This does not follow the industry guidelines or the project's coding conventions. Is the contribution polished? These are pretty important to fix, but there might be some nuances on why it is written the way it is.
- Does not match **reviewer preferences**. Suggest modifications but do not require changes, or alternatively ask for a clarification.

Minor problems, for example, typo fixes or spelling errors, can be fixed immediately by the reviewer. If the exact problem repeats, however, consider asking the original author for the fix and resubmissions; this is done to spread knowledge. You should not be making any substantive edits in the review process (barring extenuating circumstances).

Ask, don't tell. Explain your reasoning about why the code should be changed. Offer ways to improve the code. Distinguish between facts and opinions. Be aware of negative bias with the online documentation.

Responding to reviews and comments

Not always are the changes accepted on the first try. You can and will get suggestions for improvement (and other comments) from the maintainer, from the code reviewer, and from other developers. You might even get these comments in the patch form, or in a fixup commit form.

First, consider leading your response with an expression of appreciation to take time to perform a review. If anything in the review is unclear, do ask for clarification; and if there is a lack of understanding between you and the reviewer, offer clarification.

In such case, the next step is often to polish and refine changes. Then, you should resubmit them (perhaps, marking them as v2). You should respond to the review for each commit and for the whole series.

If you are responding to the comments in a pull request, reply in the same way. In the case of patch submissions via e-mail, you can put the comments for a new version (with a response to the review, or a description of the difference from the previous attempt), either between three dashes --- and the diffstat, or at the top of an e-mail separated from what is to be in the commit message by the "scissors" line, for example, ----- >8 -----. An explanation of the changes that stays constant between iterations, but nevertheless should be not included in the commit message, can be kept in the git notes (see [Chapter 8, Keeping History Clean](#)) and inserted automatically via git format-patch --notes.

Depending on the project's governance structure, you would have to wait for the changes to be considered good and ready for the inclusion. This can be the decision of a benevolent dictator for life in open-source projects, or the decision of the team leader, a committee, or a consensus. It is considered a good practice to summarize the discussion while submitting a final version of a feature.

Note that the changes that got accepted might nevertheless go through few more stages, before finally graduating to the stable branch and be present in the project.

Other recommendations

In this section, you would find the best practices and recommendations that do not fit cleanly in one of the areas described before, namely starting a project, working on a project, and integrating changes.

Don't panic, recovery is almost always possible

As long as you have committed your work, storing your changes in the repository, it will not be lost. It would only perhaps be misplaced. Git also tries to preserve your current un-committed (unsaved) work, but it cannot distinguish for example between the accidental and the conscious removing of all the changes to the working directory with `git reset --hard`. Therefore, you'd better commit or stash your current work before trying to recover lost commits.

Thanks to the reflog (both for the specific branch and for the `HEAD` ref), it is easy to undo most operations. Then, there is the list of stashed changes (see [Chapter 4, Managing Your Worktree](#)), where your changes might hide. And there is `git fsck` as the last resort. See [Chapter 11, Git Administration](#) for some further information about data recovery.

If the problem is the mess you have made of the working directory, stop and think. Do not drop your changes needlessly. With the help of interactive add, interactive reset (the `--patch` option), and interactive checkout (the same), you can usually disentangle the mess.

Running `git status` and carefully reading its output helps in many cases where you are stuck after doing some lesser-known Git operation.

If you have a problem with the rebase or merge, and you cannot pass the responsibility to another developer, there is always the third-party `git-imerge` tool.

Don't change the published history

Once you have made your changes public, you should ideally consider those revisions etched in stone, immutable and unchanging. If you find problems with commits, create a fix (perhaps, by undoing the effect of the changes with `git revert`). This is all described in [Chapter 8, *Keeping History Clean*](#).

That is, unless it is stated explicitly in the development documentation that these specific branches can be rewritten or redone; but it is nevertheless better to avoid creating such branches.

In some rare cases, you might really need to change history: remove a file, clean up a unencrypted stored password, remove accidentally added large files, and so on. If you need to do it, notify all the developers of the fact.

Numbering and tagging releases

Before you release a new version of your project, mark the version to be released with a signed tag. This ensures integrity of the just created revision.

There are various conventions on naming the release tags and on release numbering. One of more common ones is tagging releases by using, for example, `1.0.2` or `v1.0.2` as a tag name.

Note

If the integrity of the project is important, consider using signed merges for integration (that is, merging signed tags), see [Chapter 5, *Collaborative Development with Git*](#), and signed pushes, see [Chapter 11, *Git Administration*](#).

There are different conventions on naming releases. For example, with time-based releases, there is a convention of naming releases after dates, such as `2015.04` (or `15.04`). Then, there is a common convention of **semantic versioning** (<http://semver.org/>) with the `MAJOR.MINOR.PATCH` numbering, where `PATCH` increases when you are making backward-compatible bug fixes, `MINOR` is increased while adding a functionality

that is backward compatible, and the MAJOR version is increased while making incompatible API changes. Even when not using a full semantic versioning, it is common to add a third number for maintenance releases, for example, v1.0 and v1.0.3.

Automate what is possible

You should not only have the coding standards written down in the development documentation, but you also need to enforce them. Following these standards can be facilitated with client-side hooks ([Chapter 10, Customizing and Extending Git](#)), and it can be enforced with server-side hooks ([Chapter 11, Git Administration](#)).

Hooks can also help with automatically managing tickets in the issue tracker, selecting an operation based on given triggers (patterns) in the commit message. Hooks can also be used to protect against rewriting history.

Consider using third-party solutions, such as Gitolite or GitLab, to enforce rules for access control. If you need code review, use appropriate tools such as Gerrit or the pull requests of GitHub, Bitbucket, or GitLab.

Summary

These recommendations, based on the best practices of using Git as a version control system, can really help your development and your team. You have learned the steps on the road, starting from an idea, all the way ending with the changes being integrated into the project. These checklists should help you develop better code.

Appendix A. Bibliography

This Learning Path is a blend of content, all packaged up keeping your journey in mind. It includes content from the following Packt products:

- *Git Essentials*, *Ferdinando Santacroce*
- *Git Version Control Cookbook*, *Aske Olsson and Rasmus Voss*
- *Mastering Git*, *Jakub Narębski*

Index

A

- --allow-empty option
 - used, for creating empty commit / [How to do it...](#)
- --auto option / [How to do it...](#)
- 34ac2 branching point / [Branch points](#)
- a1b2c3 commit / [The first parent, Cherry picking](#)
- add -p/--patch option
 - about / [How to do it...](#)
- Agile movement techniques / [Split up features and tasks](#)
- alias, Git
 - about / [Git aliases, How to do it..., How it works...](#)
- aliases
 - examples / [More aliases, How to do it...](#)
- ancestry references
 - about / [Ancestry references](#)
 - first parent / [The first parent](#)
 - second parent / [The second parent](#)
 - / [Ancestry references](#)
- annotated tag
 - about / [Tagging commits in the repository](#)
 - / [Creating a tag](#)
- annotated tags
 - about / [Branches and tags, Annotated tags](#)
- anonymous Git protocol
 - about / [Anonymous Git protocol](#)
- Apache Subversion (SVN) / [Understanding what happens under the hood](#)
- archives
 - creating, from tree / [Creating archives from a tree, How to do it..., There's more...](#)
- Atlassian SourceTree
 - about / [Atlassian SourceTree](#)
- attribute macros

- defining / [Defining attribute macros](#)
- attributes
 - setting up, for exporting archive / [Attributes to export an archive, There's more...](#)
- author
 - versus committer / [Predefined and user defined output formats](#)
- authors
 - mapping / [Summarizing contributions](#)
- autocompletion feature
 - enabling / [Autocompletion, How it works...](#)
 - for Linux users / [Linux](#)
 - for mac users / [Mac](#)
 - on Windows / [Windows](#)
- autocorrect, Git configuration
 - about / [Autocorrect, There's more...](#)
- autocorrection
 - about / [Typos autocorrection](#)
- automatic garbage collection
 - switching off / [Turning off automatic garbage collection, How to do it...](#)
- autosquash feature, Git / [Auto-squashing commits](#)
- auto squashing
 - commits / [Auto-squashing commits , How to do it...](#)

B

- bare repositories
 - about / [Bare repositories, Bare repositories](#)
 - regular repositories, converting to / [Converting a regular repository to a bare one](#)
- bare repository
 - about / [Back up your repositories as mirror repositories](#)
 - versus mirror repository / [How to do it...](#)
- bash prompt
 - with status information / [Bash prompt with status information, How it works..., See also](#)
- basic configurations, Git

- about / [Basic configurations](#)
 - typos autocorrection / [Typos autocorrection](#)
 - push default / [Push default](#)
 - default editor, defining / [Defining the default editor](#)
- beta software branch
 - about / [The develop branch](#)
- binaries
 - storing, to another location / [Storing binaries elsewhere](#), [How to do it...](#), [How it works...](#)
- binary files
 - metadata, diffing / [Metadata diff of binary files](#), [How to do it...](#), [How it works...](#)
 - identifying / [Identifying binary files and end-of-line conversions](#)
- binary handlers
 - git-annex handler / [See also](#)
 - git-media handler / [See also](#)
 - git-bin handler / [See also](#)
- bisect command / [Debugging with git bisect](#)
- blame command
 - using / [Using the blame command](#), [How to do it...](#)
 - about / [Using the blame command](#), [There's more...](#), [Blame – the line-wise history of a file](#)
- Blob object
 - about / [Git objects](#)
- blob object
 - about / [The blob object](#)
 - writing, to database / [Writing a blob object to the database](#), [How it works...](#)
- branch
 - anatomy / [Anatomy of branches](#)
 - current branches, viewing / [Looking at the current branches](#)
 - creating / [Creating a new branch](#)
 - switching / [Switching from branch to branch](#), [Understanding what happens under the hood](#)
 - viewing / [A bird's eye view to branches](#)
 - merging / [Merging branches](#)

- merging scenarios / [Exercises, Scenario](#)
 - merged modifications / [Deal with branches' modifications](#)
 - diffing / [Diffing branches](#)
 - merge conflicts, resolving / [Resolving merge conflicts](#)
 - pushing, to remote / [Pushing a new branch to the remote](#)
 - about / [Branches and tags](#)
 - rewinding / [Discarding changes and rewinding branch](#)
- branch, rebasing
 - about / [Rebasing a branch](#)
 - merge, versus rebase / [Merge versus rebase](#)
 - types of rebase / [Types of rebase](#)
 - advanced rebasing techniques / [Advanced rebasing techniques](#)
- branch.<name>.rebase configuration / [Rebase and merge setup](#)
- branch.autosetuprebase configuration / [Rebase and merge setup](#)
- branch changes
 - discarding / [Discarding changes and rewinding branch](#)
- branch description
 - using, in commit / [Using a branch description in the commit message, How to do it...](#)
- branches
 - local branches, managing / [Managing your local branches, How to do it..., How it works..., There's more...](#)
 - remote branches / [Branches with remotes, Getting ready, There's more...](#)
 - merging, with merge commit / [Forcing a merge commit, How to do it..., There's more...](#)
 - merging, git rerere used / [Using git rerere to merge known conflicts, How to do it..., There's more...](#)
 - differentiating / [The difference between branches, How to do it...](#)
 - patches, creating from / [Creating patches from branches, How to do it..., There's more...](#)
 - creating / [Creating a new branch](#)
 - merging / [Merging a branch \(no conflicts\)](#)
 - unpublished merge, undoing / [Undoing an unpublished merge](#)
 - working with / [Working with branches](#)

- new branch, creating / [Creating a new branch](#)
- orphan branches, creating / [Creating orphan branches](#)
- selecting / [Selecting and switching to a branch](#)
- switching / [Selecting and switching to a branch](#)
- obstacles, switching to / [Obstacles to switching to a branch](#)
- anonymous branches / [Anonymous branches](#)
- Git checkout DWIM-mery / [Git checkout DWIM-mery](#)
- listing / [Listing branches](#)
- rewinding / [Rewinding or resetting a branch](#)
- resetting / [Rewinding or resetting a branch](#)
- deleting / [Deleting a branch](#)
- branch name, changing / [Changing the branch name](#)
- about / [Types and purposes of branches](#)
- types / [Types and purposes of branches](#)
- purposes / [Types and purposes of branches](#)
- long-lived branches / [Long-running, perpetual branches](#)
- short-lived branches / [Short-lived branches](#)
- workflows / [Branching workflows and release engineering](#)
- release engineering / [Branching workflows and release engineering](#)
- branches, interacting in remote repositories
 - about / [Interacting with branches in remote repositories](#)
 - upstream / [Upstream and downstream](#)
 - downstream / [Upstream and downstream](#)
 - refspec / [Remote-tracking branches and refspec, Refspec – remote to local branch mapping specification](#)
 - remote-tracking branches / [Remote-tracking branches](#)
 - fetching and pulling, versus pushing / [Fetching and pulling versus pushing](#)
 - current branches, fetching / [Pull – fetch and update current branch](#)
 - current branches, updating / [Pull – fetch and update current branch](#)
 - current branches, pushing in nonbare remote repository / [Pushing to the current branch in a nonbare remote repository](#)
 - default fetch refspec / [The default fetch refspec and push modes](#)

- push modes / [The default fetch refspec and push modes, Push modes and their use](#)
- fetching tags / [Fetching and pushing branches and tags, Fetching tags and automatic tag following](#)
- fetching branches / [Fetching branches](#)
- automatic tag following / [Fetching tags and automatic tag following](#)
- pushing branches / [Pushing branches and tags](#)
- pushing tags / [Pushing branches and tags](#)
- branches, merging
 - about / [Merging branches](#)
 - no divergence / [No divergence – fast-forward and up-to-date cases](#)
 - merge commit, creating / [Creating a merge commit](#)
 - merge strategies / [Merge strategies and their options](#)
 - merge drivers / [Reminder – merge drivers](#)
 - merges, signing / [Reminder – signing merges and merging tags](#)
 - tags, merging / [Reminder – signing merges and merging tags](#)
- branch head / [Working with branches](#)
 - rewinding / [Rewinding the branch head, softly](#)
 - resetting / [Resetting the branch head and the index](#)
- branching
 - about / [Taking another way – Git branching](#)
- branching workflow
 - Bugfix / [Deciding what to base your work on](#)
 - new feature / [Deciding what to base your work on](#)
- branch object
 - about / [The branch](#)
- branch operation
 - about / [Branches and tags](#)
- branch points
 - about / [Branch points](#)
- branch references
 - about / [Branch and tag references](#)
- bugs
 - finding, with git bisect / [Finding bugs with git bisect](#)

- built-in attributes
 - about / [Other built-in attributes](#)
- bunch of files
 - committing / [Committing a bunch of files](#)
- bundle
 - about / [Offline transport with bundles](#)
 - used, for updating existing repository / [Using bundle to update an existing repository](#)
 - utilizing / [Utilizing bundle to help with the initial clone](#)
- bundle command / [How it works...](#)

C

- --cached option
 - about / [How it works...](#)
/ [There's more...](#)
- -C option
 - about / [There's more...](#)
- caret character / [Ancestry references](#)
- carriage return (CR) character / [Formatting and whitespace](#)
- cat-file command / [How it works...](#)
 - about / [The commit object](#)
 - using / [How it works...](#)
- cat cat-me.txt command / [The blob object](#)
- centralized workflow
 - about / [The centralized workflow, Pushing to a public repository](#)
 - advantages / [The centralized workflow](#)
 - disadvantages / [The centralized workflow](#)
- centralized workflows
 - about / [Centralized workflows](#)
 - working / [How they work](#)
- changed files
 - list, obtaining of / [Getting a list of the changed files, How it works...](#)
- changes
 - searching, in revision / [Searching changes in revisions](#)
 - including / [Including, formatting, and summing up changes](#)

- formatting / [Including, formatting, and summing up changes](#)
 - summing up / [Including, formatting, and summing up changes](#)
 - stashing away / [Stashing away your changes](#)
 - integrating / [Integrating changes](#)
 - submitting / [Submitting and describing changes](#)
 - describing / [Submitting and describing changes](#)
 - peer review, completing / [The art of the change review](#)
- changes, combining
 - methods / [Methods of combining changes](#)
- changeset
 - copying / [Copying and applying a changeset](#)
 - applying / [Copying and applying a changeset](#)
 - copy, creating / [Cherry-pick – creating a copy of a changeset](#)
 - revert / [Revert – undoing an effect of a commit](#)
 - series of commits, applying from patches / [Applying a series of commits from patches](#)
 - cherry-picking / [Cherry-picking and reverting a merge](#)
 - merge, reverting / [Cherry-picking and reverting a merge](#)
- changes examining, commit
 - about / [Examining the changes to be committed](#)
 - working directory, status / [The status of the working directory](#)
 - changes to be committed / [The status of the working directory](#)
 - changes not staged for commit / [The status of the working directory](#)
 - untracked files / [The status of the working directory](#)
 - differences, examining / [Examining differences from the last revision](#)
 - unified diff output format / [Unified Git diff format](#)
- changes upstream
 - publishing / [Publishing your changes upstream](#)
- cherry picking activity / [Cherry picking](#)
- child / [Directed Acyclic Graphs](#)
- chunk
 - about / [Configuring diff output](#)
- clean command
 - about / [How to do it...](#)

- clean filter / [How to do it...](#)
- client-side hooks
 - about / [Automating Git with hooks](#)
 - commit process hooks / [Commit process hooks](#)
 - for applying patches from e-mails / [Hooks for applying patches from e-mails](#)
 - pre-rebase hook / [Other client-side hooks](#)
 - pre-push hook / [Other client-side hooks](#)
 - post-checkout hook / [Other client-side hooks](#)
 - post-merge hook / [Other client-side hooks](#)
- clone command / [How to do it...](#)
- Cmder
 - about / [Cmder](#)
 - URL, for downloading / [Cmder](#)
- code review (code collaboration) tool
 - about / [Tools to manage Git repositories](#)
- code reviews
 - about / [Submitting pull requests](#)
- collaborative workflows
 - about / [Collaborative workflows](#)
 - centralized workflow / [The centralized workflow](#)
 - peer-to-peer workflow / [The peer-to-peer or forking workflow](#)
 - integration-manager workflow / [The maintainer or integration manager workflow](#)
 - hierarchical workflow / [The hierarchical or dictator and lieutenants workflows](#)
- combined diff format / [Examining differences – the combined diff format](#)
- command line
 - about / [Git on the command line](#)
- command substitution / [Triple-dot notation](#)
- comments
 - responding to / [Responding to reviews and comments](#)
- commit
 - anatomy / [Anatomy of a commit](#)
 - author / [Author, e-mail, and date](#)

- e-mail / [Author, e-mail, and date](#)
- date / [Author, e-mail, and date](#)
- reference link / [Conclusions](#)
- tagging, in repository / [Tagging commits in the repository](#), [Getting ready](#), [How to do it...](#), [There's more...](#)
- branch description, using in / [Using a branch description in the commit message](#), [How to do it...](#)
- push, preventing / [Preventing the push of specific commits](#), [How to do it...](#), [There's more...](#)
- undoing / [Undo – remove a commit completely](#), [How to do it...](#), [How it works...](#)
- resetting / [Undo – remove a commit and retain the changes to files](#), [How to do it...](#)
- undoing, while retaining changes in staging area / [Undo – remove a commit and retain the changes in the staging area](#), [How to do it...](#), [How it works...](#)
- redo / [Redo – recreate the latest commit with new changes](#), [How to do it...](#)
- changes, reverting / [Revert – undo the changes introduced by a commit](#), [How to do it...](#), [How it works...](#)
- creating / [Creating a new commit](#)
- DAG view / [The DAG view of creating a new commit](#)
- index / [The index – a staging area for commits](#)
- changes, examining / [Examining the changes to be committed](#)
- selective commit / [Selective commit](#)
- amending / [Amending a commit](#), [Removing or amending a commit](#)
- removing / [Removing or amending a commit](#)
- splitting, with reset command / [Splitting a commit with reset](#)
- reverting / [Reverting a commit](#)
- faulty merge, reverting / [Reverting a faulty merge](#)
- reverted merges, recovering / [Recovering from reverted merges](#)
- commit-msg hook
 - about / [Commit process hooks](#)
- commit contents
 - matching / [Matching commit contents](#)

- commit hash
 - about / [The commit hash](#)
- commit message
 - external information, using / [Using external information in the commit message](#), [How to do it...](#)
- commit messages
 - about / [Commit messages](#)
 - writing, before coding / [Write commit messages before starting to code](#)
 - grepping / [Grepping the commit messages](#), [How to do it...](#)
- commit message template
 - creating / [Creating a dynamic commit message template](#), [How to do it...](#), [There's more...](#)
- commit object
 - about / [The commit object](#), [Git objects](#)
 - creating, stages / [How to do it...](#), [How it works...](#)
 - writing, to database / [Writing a commit object to the database](#), [How it works...](#)
- commit parents
 - about / [Commit parents](#)
- commit process hooks
 - about / [Commit process hooks](#)
 - pre-commit hook / [Commit process hooks](#)
 - prepare-commit-msg hook / [Commit process hooks](#)
 - commit-msg hook / [Commit process hooks](#)
 - post-commit hook / [Commit process hooks](#)
- commits
 - finding, in history / [Finding commits in history](#), [There's more...](#)
 - rebasing, to another branch / [Rebasing commits to another branch](#), [How to do it...](#), [How it works](#)
 - squashing, interactive rebase used / [Squashing commits using an interactive rebase](#), [How to do it...](#), [There's more...](#)
 - author changing, rebase used / [Changing the author of commits using a rebase](#), [How to do it...](#), [How it works...](#)
 - auto squashing / [Auto-squashing commits](#), [How to do it...](#)
 - about / [Directed Acyclic Graphs](#)

- squashing, with reset command / [Squashing commits with reset](#)
 - moving, to feature branch / [Moving commits to a feature branch](#)
- commit snapshot
 - about / [The commit snapshot](#)
- commit template
 - setting up / [Setting up and using a commit template](#), [How to do it...](#)
 - using / [Getting ready](#), [How to do it...](#)
- committer
 - versus author / [Predefined and user defined output formats](#)
- committing
 - about / [The art of committing](#)
 - right commit, building / [Building the right commit](#), [Make only one change per commit](#), [Split up features and tasks](#), [Write commit messages before starting to code](#), [Include the whole change in one commit](#), [Don't be afraid to commit](#), [Isolate meaningless commits](#), [Adding bulleted details lines, when needed](#), [Conclusions](#)
- commit tools
 - about / [Types of graphical tools](#)
- commit walker downloader / [Local transport](#)
- Common Vulnerabilities and Exposures (CVE) / [Matching commit contents](#)
 - about / [Writing a good commit message](#)
- configuration, Git
 - querying / [Querying the existing configuration](#), [How it works...](#), [There's more...](#)
 - rebase and merge setup / [Getting ready](#), [Rebase and merge setup](#)
 - object expiry / [Expiry of objects](#)
 - autocorrect / [Autocorrect](#), [There's more...](#)
- configuration, GLOBAL layer / [Configuration targets](#), [How to do it...](#), [There's more...](#)
- configuration, LOCAL layer / [Configuration targets](#), [How to do it...](#), [There's more...](#)
- configuration, SYSTEM layer / [Configuration targets](#), [How to do it...](#)

- configuration architecture, Git / [Configuration architecture](#)
- configuration files, Git
 - editing, manually / [Editing configuration files manually](#)
- configuration levels, Git
 - about / [Configuration levels](#)
 - system level / [System level](#)
 - global level / [Global level](#)
 - repository level / [Repository level](#)
- configurations, Git
 - listing / [Listing configurations](#)
- configuration variables, Git
 - reference link / [Other configurations](#)
- conflicts
 - resolving, by removing file / [Resolving conflicts by removing the file](#)
- content-addressed storage / [Content-addressed storage](#)
- Continuous Delivery structure / [The GitHub flow](#)
- contributions
 - summarizing / [Summarizing contributions](#)
- Coordinated Universal Time (UTC)
 - about / [Git objects](#)
- credential helpers
 - about / [Credential helpers](#)
- credentials
 - about / [Credentials/password management](#)
- credentials/password management
 - about / [Credentials/password management](#)
 - asking for password / [Asking for passwords](#)
 - public key authentication, for SSH / [Public key authentication for SSH](#)
 - credential helpers / [Credential helpers](#)
- credentials helpers
 - about / [Credential helpers and remote helpers](#)
- current push mode
 - about / [The current push mode for the blessed repository workflow](#)

- current variable / [Getting ready](#)

D

- --diff-filter=DA option / [How to do it...](#)
- d0dd1f61 commit / [Tracing changes in a file](#)
- DAG
 - about / [Viewing the DAG](#)
 - viewing / [How to do it..., How it works...](#)
- dangling commit / [How it works...](#)
- database
 - blob object, writing to / [Writing a blob object to the database, How it works...](#)
 - tree object, writing to / [Writing a tree object to the database, How it works...](#)
 - commit object, writing to / [Writing a commit object to the database, How it works...](#)
- data model, Git / [Introduction](#)
- data recovery
 - about / [Data recovery and troubleshooting](#)
 - lost commit, recovering / [Recovering a lost commit](#)
- default branch
 - setting, of remote / [Setting the default branch of remote](#)
- descriptioInCommit branch
 - about / [Getting ready](#)
- detached HEAD situation
 - about / [Branches and tags](#)
- develop branch, GitFlow / [The develop branch](#)
- development workflows
 - augmenting / [Augmenting development workflows](#)
- diff
 - revision range notation, using in / [Triple-dot notation](#)
- diff-tree command
 - using / [How to do it...](#)
- diff command
 - about / [There's more...](#)
- diff configuration

- about / [Diff and merge configuration](#)
- diffing binary files / [Generating diffs and binary files](#)
- diff output
 - configuring / [Configuring diff output](#)
- diff utility
 - URL / [Diffing branches](#)
- Directed Acyclic Graph (DAG)
 - about / [Directed Acyclic Graphs](#)
- directed edges / [Directed Acyclic Graphs](#)
- directed graph
 - about / [Directed Acyclic Graphs](#)
- directories
 - examining / [Examining files and directories](#)
- distributed version control systems / [An introduction to version control and Git](#)
- distributed version control systems (DVCS)
 - about / [Directed Acyclic Graphs](#)
- double dot notation / [Double dot notation](#)
- do what I mean (DWIM)
 - about / [Available filter types for filter-branch and their use](#)
- Do What I Mean (DWIM) / [Git checkout DWIM-tery](#)
- downloaded remote changes
 - applying / [Applying downloaded changes](#)
- dumb protocols
 - about / [Dumb protocols](#)

E

- Easy Git (eg) / [Alternative command line](#)
- end-of-line conversions
 - identifying / [Identifying binary files and end-of-line conversions](#)
- end of line (eol) characters / [Identifying binary files and end-of-line conversions](#)
- env-filter
 - about / [There's more...](#)
- environment configurations, Git
 - about / [Setting up other environment configurations](#)

- environment variables
 - affecting global behavior / [Environment variables affecting global behavior](#)
 - affecting repository locations / [Environment variables affecting repository locations](#)
 - affecting committing / [Environment variables affecting committing](#)
- exif-diff filter / [How to do it...](#)
- exiftool diff filter
 - setting up / [How to do it...](#)
- external information
 - using, in commit message / [Using external information in the commit message](#), [How to do it...](#)
- external tools
 - for large-scale history rewriting / [External tools for large-scale history rewriting](#)
- eXtreme Programming
 - URL / [Submitting pull requests](#)

F

- --format=<format> option / [There's more...](#)
- failed merges
 - examining / [Examining failed merges](#)
 - conflict markers, in worktree / [Conflict markers in the worktree](#)
 - three stages, in index / [Three stages in the index](#)
 - differences, examining / [Examining differences – the combined diff format](#)
 - git log / [How do we get there: git log --merge](#)
- feature branch
 - commits, moving to / [Moving commits to a feature branch](#)
 - about / [Working on a topic branch](#)
- feature branches, GitFlow / [The feature branches](#)
- feature branch workflow
 - about / [Feature branch workflow](#)
- file
 - unstaging / [Unstaging a file](#), [Unstaging a file](#)

- committing to Subversion, with Git as client / [Committing a file to Subversion using Git as a client](#)
 - sensitive data, removing from / [Rewriting history – changing a single file](#), [How it works...](#)
 - adding / [Interactive add](#), [How to do it...](#), [There's more...](#)
- file attributes
 - checking / [Checking the attributes of a file](#), [How to do it...](#)
 - about / [File attributes](#)
- file contents
 - searching / [Searching file contents](#)
- file history
 - about / [History of a file](#)
 - path limiting / [Path limiting](#)
 - simplification / [History simplification](#)
- file identity / [Blame – the line-wise history of a file](#)
- file information
 - displaying / [Displaying the file information](#), [How to do it...](#), [There's more...](#)
- file manager integration (or graphical shell integration)
 - about / [Types of graphical tools](#)
- files
 - ignoring / [Ignoring some files and folders by default](#), [Ignoring files](#), [How to do it...](#), [There's more...](#), [See also...](#), [Ignoring files](#), [Which types of file should be ignored?](#)
 - viewing, at revision / [Viewing a revision and a file at revision](#)
 - un-tracking / [Ignoring files](#), [Un-tracking, un-staging, and un-modifying files](#)
 - re-tracking / [Ignoring files](#)
 - marking, as intentionally untracked (ignored) / [Marking files as intentionally untracked](#)
 - transforming / [Transforming files \(content filtering\)](#)
 - examining / [Examining files and directories](#)
 - un-staging / [Un-tracking, un-staging, and un-modifying files](#)
 - un-modifying / [Un-tracking, un-staging, and un-modifying files](#)
 - resetting, to old version / [Resetting a file to the old version](#)
 - removing from history, with BFG Repo Cleaner / [Removing](#)

[files from the history with BFG Repo Cleaner](#)

- file statuses
 - about / [File statuses](#)
 - untracked / [File statuses](#)
 - unmodified / [File statuses](#)
 - staged / [File statuses](#)
 - modified / [File statuses](#)
- file status life cycle
 - about / [Repository structure and file status life cycle](#)
- Filesystem Hierarchy Standard (FHS) / [Git configuration files](#)
- filter-branch
 - running, without filters / [Running the filter-branch without filters](#)
- filter-branch command / [Introduction](#)
- filters
 - about / [Scripted rewrite with the git filter-branch](#)
- filter types
 - used, for filter-branch / [Available filter types for filter-branch and their use](#)
 - using / [Available filter types for filter-branch and their use](#)
- fixed issues
 - extracting / [Extracting fixed issues, How to do it...](#)
- folders
 - ignoring / [Ignoring some files and folders by default](#)
- foreign SCM repositories
 - using, as remotes / [Using foreign SCM repositories as remotes](#)
- forking
 - about / [Forking a repository](#)
- forks
 - about / [Forking a repository](#)
- fsck command
 - used, for searching lost changes / [Finding lost changes with git fsck, Getting ready, How to do it..., How it works...](#)
 - about / [Finding lost changes with git fsck](#)

G

- --global
 - about / [How to do it...](#)
- .gitignore
 - about / [Ignoring files](#)
- .gitignore file
 - URL, for syntax / [Ignoring some files and folders by default](#)
- .gitignore templates
 - reference link / [Which types of file should be ignored?](#)
- garbage collection
 - manually running / [Running garbage collection manually](#), [How to do it...](#), [How it works...](#)
- gc.pruneexpire / [Expiry of objects](#)
- gc.reflogexpire / [Expiry of objects](#)
- gc.reflogexpireunreachable / [Expiry of objects](#)
- Gerrit
 - about / [Other categories and uses of notes](#)
- Gerrit Code Review
 - about / [How it works...](#)
- Git
 - about / [Installing Git](#), [Working with remotes](#), [Branches and tags](#)
 - installing / [Installing Git](#)
 - references, for downloading / [Installing Git](#)
 - configuration levels / [Configuration levels](#), [System level](#)
 - used, for working on Subversion repository / [Working on a Subversion repository using Git](#)
 - Subversion repository, cloning from / [Cloning a Subversion repository from Git](#)
 - using, with Subversion repository / [Using Git with a Subversion repository](#)
 - learning, in visual manner / [Learning Git in a visual manner](#)
 - data model / [Introduction](#)
 - objects / [Git's objects](#), [Getting ready](#)
 - three stages, for creating commit object / [How to do it...](#), [How it works...](#)
 - existing configuration, querying / [Querying the existing configuration](#), [How it works...](#), [There's more...](#)

- alias / [Git aliases, How to do it..., How it works...](#)
 - clean filter / [How to do it...](#)
 - smudge filter / [How to do it...](#)
 - environment variables, using / [Environment variables used by Git](#)
 - troubleshooting / [Troubleshooting Git](#)
- Git, automating
 - about / [Automating Git with hooks](#)
 - Git hook, installing / [Installing a Git hook](#)
 - template, for repositories / [A template for repositories](#)
 - client-side hooks / [Client-side hooks](#)
 - server-side hooks / [Server-side hooks](#)
- Git, extending
 - about / [Extending Git](#)
 - command aliases / [Command aliases for Git](#)
 - new commands, adding / [Adding new Git commands](#)
 - remote helpers / [Credential helpers and remote helpers](#)
 - credentials helpers / [Credential helpers and remote helpers](#)
- Git, on command line
 - about / [Git on the command line](#)
 - Git-aware command prompt / [Git-aware command prompt](#)
 - command-line completion, for Git / [Command-line completion for Git](#)
 - auto correction, for Git commands / [Autocorrection for Git commands](#)
 - command line, customizing / [Making the command line prettier](#)
 - alternative porcelain / [Alternative command line](#)
- Git, on Internet
 - about / [Git on the Internet](#)
 - Git community, on Google+ / [Git community on Google+](#)
 - GitMinutes / [GitMinutes and Thomas Ferris Nicolaisen's blog](#)
 - Thomas Ferris Nicolaisen blog / [GitMinutes and Thomas Ferris Nicolaisen's blog](#)
 - Ferdinando Santacroce's blog / [Ferdinando Santacroce's blog](#)
- git-annex handler
 - URL / [See also](#)

- git-bin handler
 - URL / [See also](#)
- Git-enforced policy
 - implementing, hooks used / [Using hooks to implement the Git-enforced policy](#)
 - enforcing, with server-side hooks / [Enforcing the policy with server-side hooks](#)
 - policy violations, notifying with client-side hooks / [Early notices about policy violations with client-side hooks](#)
- Git-flow
 - about / [Git-flow – a successful Git branching model](#)
- git-media handler
 - URL / [See also](#)
- git-media tool
 - reference link / [Obligatory file transformations](#)
- Git actions
 - viewing, with git reflog / [Viewing past Git actions with git reflog](#), [How to do it...](#), [How it works...](#)
- git add command / [How to do it...](#), [Interactive add](#)
- Git aliases / [Typing is boring – Git aliases](#)
 - about / [Git aliases](#)
 - shortcut, to common commands / [Shortcuts to common commands](#)
 - commands, creating / [Creating commands](#)
 - git unstage / [git unstage](#)
 - git undo / [git undo](#)
 - git last / [git last](#)
 - git difflast / [git difflast](#)
 - advanced aliases, with external commands / [Advanced aliases with external commands](#)
 - removing / [Removing an alias](#)
 - git command, aliasing / [Aliasing the git command itself](#)
 - configuring, with git config / [How to do it...](#)
 - using / [How to do it...](#), [How it works...](#)
- git am command / [How it works...](#), [There's more...](#)
- git apply command / [There's more...](#)

- git archive command / [Archiving the repository](#)
- gitattributes / [Unified Git diff format](#)
- Git attributes file
 - about / [Per-file configuration with gitattributes](#)
- git bisect
 - bugs, finding with / [Finding bugs with git bisect](#)
- git bisect command
 - about / [Debugging with git bisect](#)
 - debugging with / [Debugging with git bisect](#), [Getting ready](#), [How to do it...](#), [There's more...](#)
- git blame command / [Tracing changes in a file](#)
- git branch command
 - about / [There's more...](#)
- Git bundle
 - using / [Using a Git bundle](#), [How to do it...](#), [There's more...](#)
- git bundle command / [Bundling the repository](#), [Creating Git bundles](#)
- Git bundles
 - creating / [Creating Git bundles](#), [How to do it...](#)
- Git by example / [Git by example](#)
- git cat-file -p command / [How it works...](#)
- git cherry-pick command / [Cherry picking](#)
- git clone command / [Converting a regular repository to a bare one](#)
- git command / [Sending patches](#)
- Git command, running
 - about / [Running our first Git command](#)
 - new repository, setting up / [Setting up a new repository](#)
 - text file, creating / [Adding a file](#)
 - added file, committing / [Commit the added file](#)
 - committed file, modifying / [Modify a committed file](#)
- Git commands
 - versus Subversion commands / [Comparing Git and Subversion commands](#)
- git commit --amend --reset-author / [How to do it...](#)
- Git community, on Google+
 - URL / [Git community on Google+](#)
- git config

- used, for configuring Git aliases / [How to do it...](#)
- git config command / [Configuration architecture](#)
 - about / [How it works...](#)
- git config file
 - about / [How it works...](#)
- git config rebase.autosquash true / [How to do it...](#)
- Git configuration
 - dissecting / [Dissecting the Git configuration](#)
 - architecture / [Configuration architecture](#)
 - about / [Configuring Git](#)
 - command-line options / [Command-line options and environment variables](#)
 - environment variables / [Command-line options and environment variables](#)
 - configuration files / [Git configuration files](#)
 - configuration files syntax / [The syntax of Git configuration files](#)
 - accessing / [Accessing the Git configuration](#)
 - basic client-side configuration / [Basic client-side configuration](#)
 - merge / [The rebase and merge setup, configuring pull](#)
 - pull, configuring / [The rebase and merge setup, configuring pull](#)
 - undo information, preserving / [Preserving undo information – the expiry of objects](#)
 - formatting / [Formatting and whitespace](#)
 - whitespace / [Formatting and whitespace](#)
 - server-side configuration / [Server-side configuration](#)
 - per-file configuration, with gitattributes / [Per-file configuration with gitattributes](#)
- git describe command / [How to do it...](#)
- git diff command
 - about / [How it works...](#)
- git difflast alias
 - about / [git difflast](#)
- git directory
 - about / [The working directory](#)
- git fetch command / [Checking for modifications and downloading them](#)

- about / [There's more...](#)
- git filter-branch
 - using / [Examples of using the git filter-branch](#)
- git filter-branch command
 - about / [How it works...](#)
 - env-filter / [There's more...](#)
 - tree-filter / [There's more...](#)
 - msg-filter / [There's more...](#)
 - subdirectory-filter / [There's more...](#)
 - filter types, defining / [Available filter types for filter-branch and their use](#)
- GitFlow
 - about / [GitFlow](#)
 - reference link, for blog post / [GitFlow](#)
 - master branch / [The master branch](#)
 - hotfixes branches / [Hotfixes branches](#)
 - develop branch / [The develop branch](#)
 - release branch / [The release branch](#)
 - feature branches / [The feature branches](#)
 - conclusions / [Conclusion](#)
- GitFlow commands
 - reference link / [Conclusion](#)
- git format-patch command / [There's more...](#)
- git gc command / [Running garbage collection manually, How it works...](#)
- Git GUI
 - about / [Git GUI](#)
 - interactive add, using with / [Interactive add with Git GUI, How to do it...](#)
- Git GUI, for Linux
 - URL / [Linux](#)
- Git GUI clients
 - about / [Git GUI clients](#)
 - Windows / [Windows](#)
 - Mac OS X / [Mac OS X](#)
 - Linux / [Linux](#)

- Git hook
 - installing / [Installing a Git hook](#)
- Git hosting solutions
 - about / [Tools to manage Git repositories](#)
- GitHub
 - about / [Working with remotes](#)
 - URL / [Setting up a new GitHub account](#)
 - local repository, publishing to / [Going backward: publish a local repository to GitHub](#)
- GitHub account
 - setting up / [Setting up a new GitHub account](#)
- GitHub flow
 - about / [The GitHub flow](#)
 - reference link / [The GitHub flow, Conclusions](#)
 - rules / [The GitHub flow, Pushing to named branches constantly, Conclusions](#)
- GitHub for Windows / [GitHub for Windows](#)
- gitignore files
 - about / [Marking files as intentionally untracked](#)
- gitignore templates
 - URL / [Choosing which files to keep under version control](#)
- Git internals
 - defining / [An introduction to Git internals](#)
 - Git objects / [Git objects](#)
 - plumbing command / [The plumbing and porcelain Git commands](#)
 - porcelain command / [The plumbing and porcelain Git commands](#)
- Gitk
 - history, viewing with / [Viewing history with Gitk, How it works...](#)
- git last alias
 - about / [git last](#)
- git log command / [Viewing the history, Viewing the DAG, Searching through history code, Extracting the top contributor](#)
 - about / [There's more...](#)

- git ls-tree command / [There's more...](#)
- git merge command / [Cherry picking](#)
- GitMinutes
 - URL / [GitMinutes and Thomas Ferris Nicolaisen's blog](#)
- git push command / [What do I send to the remote when I push?](#),
[Push default](#)
- Git Queues(gq) / [Alternative command line](#)
- Git Quilt (Guilt)
 - about / [External tools – patch management interfaces](#)
- git rebase --interactive
 - running / [How to do it...](#)
- git rebase -continue
 - used, for continuing rebase / [Continuing a rebase with merge conflicts](#), [How to do it](#), [How it works](#)
- git rebase -i command / [How to do it...](#)
- Git references
 - about / [Git references](#)
 - symbolic references / [Symbolic references](#)
 - ancestry references / [Ancestry references](#)
- Git remote
 - about / [Working with remotes](#)
- Git repositories
 - managing / [Managing large Git repositories](#)
 - hosting / [Git on the server](#)
 - server-side hooks / [Server-side hooks](#)
 - hooks, for implementing Git-enforced policy / [Using hooks to implement the Git-enforced policy](#)
 - signed pushes / [Signed pushes](#)
 - serving / [Serving Git repositories](#)
 - local protocol / [Local protocol](#)
 - SSH protocol / [SSH protocol](#)
 - anonymous Git protocol / [Anonymous Git protocol](#)
 - smart HTTP(S) protocol / [Smart HTTP\(S\) protocol](#)
 - dumb protocols / [Dumb protocols](#)
 - remote helpers / [Remote helpers](#)
 - tools, for managing / [Tools to manage Git repositories](#)

- tips and tricks, for hosting / [Tips and tricks for hosting repositories](#)
- Git repositories, handling with large binary files
 - about / [Handling repositories with large binary files](#)
 - binary asset folder, splitting into separate submodule / [Splitting the binary asset folder into a separate submodule](#)
 - large binary files, storing outside repository / [Storing large binary files outside the repository](#)
- Git repositories, handling with long history
 - about / [Handling repositories with a very long history](#)
 - shallow clones, using / [Using shallow clones to get truncated history](#)
 - single branch, cloning / [Cloning only a single branch](#)
- Git repository management solutions
 - about / [Tools to manage Git repositories](#)
- git rerere
 - used, for merging branches / [Using git rerere to merge known conflicts](#), [How to do it...](#), [There's more...](#)
 - about / [Using git rerere to merge known conflicts](#)
- git reset command / [Unstaging a file](#)
- Git scripts
 - configuring / [Configuring and using Git scripts](#), [How to do it...](#)
 - using / [How to do it...](#)
- git send-email command / [Sending patches](#), [There's more...](#)
- git show command / [Tracing changes in a file](#), [There's more...](#)
 - about / [There's more...](#)
- git stash
 - using / [Using git stash](#), [Getting ready](#), [How to do it...](#), [How it works...](#), [There's more...](#), [Using git stash](#)
- git stash command / [Using git stash](#), [There's more...](#)
- git stash list command / [How to do it...](#)
- git status command
 - about / [There's more...](#)
- Git submodules solution
 - git submodule command / [Gitlinks, .git files, and the git submodule command](#)

- .git files / [Gitlinks, .git files, and the git submodule command](#)
- submodule, adding as submodule / [Adding a subproject as a submodule](#)
- superprojects, cloning with submodules / [Cloning superprojects with submodules](#)
- submodules, updating after superproject changes / [Updating submodules after superproject changes](#)
- changes, examining in submodule / [Examining changes in a submodule](#)
- updates, from upstream of submodule / [Getting updates from the upstream of the submodule](#)
- submodule changes upstream, sending / [Sending submodule changes upstream](#)
- git subtree command / [See also](#)
- git tag command
 - about / [How to do it...](#)
- Git tags / [Highlighting an important commit – Git tags](#)
 - Subversion tags, converting to / [Converting Subversion tags to Git tags](#)
- git undo alias
 - about / [git undo](#)
- git unstage alias
 - about / [git unstage](#)
- GIT_PS1_DESCRIBE_STYLE variable
 - about / [How it works...](#)
- GIT_PS1_SHOWDIRTYSTATE variable
 - about / [How it works...](#)
- GIT_PS1_SHOWSTASHSTATE variable
 - about / [How it works...](#)
- GIT_PS1_SHOWUNTRACKEDFILES variable
 - about / [How it works...](#)
- GIT_PS1_SHOWUPSTREAM variable
 - about / [How it works...](#)
- GLOBAL layer
 - configuring / [Configuration targets, How to do it..., There's more...](#)

- GNU Portability Library (Gnulib) / [Performing a 3-way merge](#)
- Gradle / [Submodule versus subtree merging](#)
- graduation branches workflow
 - about / [The graduation, or progressive-stability branches workflow](#)
- graphical blame
 - about / [Types of graphical tools](#)
- graphical history viewer
 - about / [Types of graphical tools](#)
- graphical interface, Git
 - about / [Graphical interfaces](#)
 - graphical tools / [Types of graphical tools](#)
 - graphical diff tools / [Graphical diff and merge tools](#)
 - graphical merge tools / [Graphical diff and merge tools](#)
 - examples / [Graphical interface examples](#)
- graphical tools
 - graphical history viewer / [Types of graphical tools](#)
 - graphical blame / [Types of graphical tools](#)
 - commit tools / [Types of graphical tools](#)

H

- hard reset / [Undo – working with a dirty area](#)
- hash-object command
 - using / [How to do it..., How it works...](#)
- HEAD command
 - about / [HEAD – the implicit revision](#)
 - FETCH_HEAD / [HEAD – the implicit revision](#)
 - ORIG_HEAD / [HEAD – the implicit revision](#)
- HEAD pointer
 - about / [Branches and tags](#)
- helpers mechanism
 - about / [Credential helpers and remote helpers](#)
- hierarchical workflow
 - about / [The hierarchical or dictator and lieutenants workflows](#)
 - advantages / [The hierarchical or dictator and lieutenants workflows](#)

- disadvantages / [The hierarchical or dictator and lieutenants workflows](#)
- history
 - viewing, with Gitk / [Viewing history with Gitk](#), [How it works...](#)
 - commits, finding in / [Finding commits in history](#), [There's more...](#)
 - searching / [Searching history](#)
 - rewriting / [Rewriting history](#), [Rewriting history and notes](#)
 - last commit, amending / [Amending the last commit](#)
 - interactive rebase / [An interactive rebase](#)
- history, viewing of repository
 - about / [Viewing the history](#)
 - commit, anatomy / [Anatomy of a commit](#)
 - bunch of files, committing / [Committing a bunch of files](#)
 - important commit, highlighting / [Highlighting an important commit – Git tags](#)
- history, without rewriting
 - amending / [Amending history without rewriting](#)
- history code
 - searching through / [Searching through history code](#), [How it works...](#), [There's more...](#)
- hook
 - about / [Automating Git with hooks](#)
 - client-side hooks / [Automating Git with hooks](#)
 - server-side hooks / [Automating Git with hooks](#)
 - pre hooks / [Automating Git with hooks](#)
 - post hooks / [Automating Git with hooks](#)
- hotfixes branches, GitFlow / [Hotfixes branches](#)

I

- --ignore-unmatch option
 - about / [How it works...](#)
- --include-untracked option / [There's more...](#)
- --index-filter option / [How to do it...](#)
- --index option / [There's more...](#)
- -i option
 - about / [How to do it...](#)

- IDE integration
 - about / [Types of graphical tools](#)
- ignored files
 - displaying / [Showing and cleaning ignored files](#), [There's more...](#)
 - cleaning / [Showing and cleaning ignored files](#), [There's more...](#)
 - examining / [Listing ignored files](#), [Ignoring ignored filesexamining changes in tracked files](#)
- ignore patterns / [Adding files in bulk and removing files](#)
- index / [The index – a staging area for commits](#)
 - about / [File statuses](#), [Ignoring files](#)
 - resetting / [Resetting the branch head and the index](#)
- index on master / [How it works...](#)
- information
 - updating, of remotes / [Updating information about remotes](#)
- installation, Git / [Installing Git](#)
- integration branches / [The graduation, or progressive-stability branches workflow](#)
- integration manager workflow
 - about / [The maintainer or integration manager workflow](#)
 - advantages / [The maintainer or integration manager workflow](#)
 - disadvantages / [The maintainer or integration manager workflow](#)
- interactive add
 - starting / [Interactive add](#), [How to do it...](#), [There's more...](#)
 - with Git GUI / [Interactive add with Git GUI](#), [How to do it...](#)
- interactive rebase / [Amending a commit](#), [Squashing commits with reset](#)
 - used, for squashing commits / [Squashing commits using an interactive rebase](#), [How to do it...](#), [There's more...](#)
 - about / [An interactive rebase](#)
 - commits, reordering / [Reordering, removing, and fixing commits](#)
 - commits, removing / [Reordering, removing, and fixing commits](#)
 - commits, fixing / [Reordering, removing, and fixing commits](#)
 - commits, squashing / [Squashing commits](#)
 - commits, splitting / [Splitting commits](#)
 - rebased commit, testing / [Testing each rebased commit](#)

J

- JGit project
 - about / [Extracting fixed issues](#)
- JGit repository
 - cloning / [Getting ready](#)
 - splitting / [Getting ready](#), [How to do it...](#), [How it works...](#), [There's more...](#)
- jgit repository
 - cloning / [Getting ready](#)

K

- keyword expansion
 - filters, creating with attribute functionality / [Keyword expansion with attribute filters](#), [How to do it...](#), [How it works...](#), [There's more...](#)
 - / [Keyword expansion and substitution](#)
- keyword substitution / [Keyword expansion and substitution](#)

L

- -l option / [How to do it...](#)
- layers
 - LOCAL layer / [Configuration targets](#)
 - SYSTEM layer / [Configuration targets](#)
 - GLOBAL layer / [Configuration targets](#)
- leaf nodes / [Directed Acyclic Graphs](#)
- LearnGitBranching
 - URL / [Learning Git in a visual manner](#)
- lib_a file
 - about / [How to do it...](#)
- lib_a_master branch / [How to do it...](#)
- lieutenants
 - about / [The hierarchical or dictator and lieutenants workflows](#)
- lightweight tag
 - about / [Tagging commits in the repository](#), [Lightweight tags](#)
- line-ending / [Formatting and whitespace](#)

- Linus branch
 - about / [The Linux kernel workflow](#)
- Linux
 - about / [Linux](#)
- Linux kernel workflow
 - about / [The Linux kernel workflow](#)
- list
 - obtaining, of changed files / [Getting a list of the changed files, How it works...](#)
- list of branches
 - modifying, tracked by node / [Changing the list of branches tracked by remote](#)
- local branch
 - pushing, to remote repository / [Pushing a local branch to a remote repository](#)
- local branches
 - managing / [Managing your local branches, How to do it..., How it works..., There's more...](#)
- LOCAL layer
 - configuring / [Configuration targets, How to do it..., There's more...](#)
- local protocol
 - about / [Local protocol](#)
- local repository
 - publishing, to GitHub / [Going backward: publish a local repository to GitHub](#)
 - remote, adding to / [Adding a remote to a local repository](#)
- local Subversion repository
 - creating / [Creating a local Subversion repository](#)
- local Subversion server
 - setting up / [Setting up a local Subversion server](#)
- local transport
 - about / [Local transport](#)
- long-lived branches
 - integration / [Integration, graduation, or progressive-stability branches](#)

- graduation / [Integration, graduation, or progressive-stability branches](#)
- progressive-stability / [Integration, graduation, or progressive-stability branches](#)
- per-release branches / [Per-release branches and per-release maintenance](#)
- per-release maintenance / [Per-release branches and per-release maintenance](#)
- hotfix branches / [Hotfix branches for security fixes](#)
- per-deployment branches / [Per-customer or per-deployment branches](#)
- per-customer branches / [Per-customer or per-deployment branches](#)
- automation branches / [Automation branches](#)
- mob branches, for anonymous push access / [Mob branches for anonymous push access](#)
- orphan branch trick / [The orphan branch trick](#)
- lost changes
 - searching, with git fsck / [Finding lost changes with git fsck](#), [Getting ready](#), [How to do it...](#), [How it works...](#)
- ls-files command
 - using / [Displaying the file information](#), [How to do it...](#), [There's more...](#)
- ls-tree command
 - using / [How to do it...](#)

M

- -M option
 - about / [There's more...](#)
- Mac OS X
 - about / [Mac OS X](#)
- main branches / [GitFlow](#)
- maint branch / [Branch points](#)
- management of remotes
 - about / [Types of graphical tools](#)
- man pages

- about / [The staging area](#)
- Markdown
 - URL / [Working with repositories](#), [Setting up a new GitHub account](#)
- master branch / [How to do it...](#), [Branch points](#)
 - about / [How to do it...](#)
- master branch, GitFlow / [The master branch](#)
- matching push mode
 - about / [The matching push mode for maintainers](#)
- Maven / [Submodule versus subtree merging](#)
- meaningless commits
 - isolating / [Isolate meaningless commits](#)
- merge
 - undoing / [Undoing a merge or a pull](#)
- merge commit
 - forcing, for merging branches / [Forcing a merge commit](#), [Getting ready](#), [How to do it...](#), [There's more...](#)
- merge commits
 - reverting / [Reverting a merge](#), [Getting ready](#), [How to do it...](#), [There is more...](#)
 - about / [Merge commits](#)
- merge configuration
 - about / [Diff and merge configuration](#)
- merge conflict, resolving
 - about / [Resolving a merge conflict](#)
 - files, adding in bulk / [Adding files in bulk and removing files](#)
 - files, removing / [Adding files in bulk and removing files](#)
 - file changes, undoing / [Undoing changes to a file](#)
- merge conflicts
 - resolving / [Resolving merge conflicts](#), [Resolving merge conflicts](#)
 - collisions, editing / [Edit collisions](#)
 - about / [Resolving merge conflicts](#)
 - three-way merge / [The three-way merge](#)
 - failed merges, examining / [Examining failed merges](#)
 - avoiding / [Avoiding merge conflicts](#)
 - merge options / [Useful merge options](#)

- rerere (reuse recorded resolutions) / [Rerere – reuse recorded resolutions](#)
- dealing with / [Dealing with merge conflicts](#)
- merge, aborting / [Aborting a merge](#)
- version, selecting / [Selecting ours or theirs version](#)
- manual file remerging / [Scriptable fixes – manual file remerging](#)
- scriptable fixes / [Scriptable fixes – manual file remerging](#)
- graphical merge tools, using / [Using graphical merge tools](#)
- files, marking as resolved / [Marking files as resolved and finalizing merges](#)
- merges, finalizing / [Marking files as resolved and finalizing merges](#)
- rebase conflicts, resolving / [Resolving rebase conflicts](#)
- git-imerge / [git-imerge – incremental merge and rebase for git](#)
- merging, branches
 - with merge commit / [Forcing a merge commit, Getting ready, How to do it..., There's more...](#)
 - with git rerere / [Using git rerere to merge known conflicts, How to do it..., There's more...](#)
- migrating, to Git
 - prerequisites / [Prerequisites](#)
- mirror repository
 - about / [Back up your repositories as mirror repositories](#)
 - versus bare repository / [How to do it...](#)
- mixed reset / [Resetting the branch head and the index](#)
- modification
 - uploading, to remotes / [Uploading modifications to remotes](#)
- msg-filter
 - about / [There's more...](#)
- Msysgit installation
 - about / [Windows](#)
- MUA (Mail User Agent) / [How it works...](#)
- multiple working directories / [Multiple working directories](#)

N

- --name-status option / [How to do it...](#)

- --no-commit option / [How to do it...](#)
- -<n> option / [There's more...](#)
- -n, --numbered option / [There's more...](#)
- native Git protocol
 - about / [Native Git protocol](#)
- nodes
 - about / [Directed Acyclic Graphs](#)
- nodes, DAG
 - root nodes / [Directed Acyclic Graphs](#)
 - leaf nodes / [Directed Acyclic Graphs](#)
- notes
 - about / [Introduction](#)
 - adding / [Adding your first Git note](#), [Getting ready](#), [How to do it...](#), [There's more...](#)
 - sorting, by category / [Separating notes by category](#), [Getting ready](#), [How to do it...](#), [How it works...](#)
 - retrieving, from remote repository / [Retrieving notes from the remote repository](#), [Getting ready](#), [How to do it...](#), [How it works...](#)
 - pushing, to repository / [Pushing notes to a remote repository](#), [How to do it...](#), [There's more...](#)
 - additional information, storing with / [Storing additional information with notes](#)
 - adding, to commit / [Adding notes to a commit](#)
 - storing / [How notes are stored](#)
 - using / [Other categories and uses of notes](#)
 - rewriting / [Rewriting history and notes](#)
 - publishing / [Publishing and retrieving notes](#)
 - retrieving / [Publishing and retrieving notes](#)
 - fetching / [Publishing and retrieving notes](#)
- number of revisions
 - limiting / [Limiting the number of revisions](#)

O

- -1 option
 - about / [How it works...](#)

- -o <dir> option / [There's more...](#)
- -o latest-commit option
 - about / [How it works...](#)
- -o option / [There's more...](#)
- object expiry, Git configuration
 - about / [Expiry of objects](#)
 - gc.reflogexpire / [Expiry of objects](#)
 - gc.reflogexpireunreachable / [Expiry of objects](#)
 - gc.pruneexpire / [Expiry of objects](#)
- objects, Git
 - about / [Git's objects](#)
 - commit / [The commit object](#)
 - tree / [The tree object](#)
 - blob / [The blob object](#)
 - branch / [The branch](#)
 - tag / [The tag object](#)
- obligatory file transformations
 - about / [Obligatory file transformations](#)
- offline transport, with bundles
 - about / [Offline transport with bundles](#)
- oh-my-zsh
 - URL / [See also](#)
- old version
 - files, resetting to / [Resetting a file to the old version](#)
- OpenDocument Format (ODF) files / [Transforming files \(content filtering\)](#)
- origin alias
 - about / [The origin](#)
- orphan branches / [Directed Acyclic Graphs](#)
- output
 - selecting / [Selecting and formatting the git log output](#)
 - formatting / [Selecting and formatting the git log output](#)

P

- --prefix option / [How to do it...](#)
- --preserve-merges flag / [How to do it...](#)

- --pretty=fuller option / [How it works...](#)
- parent / [Directed Acyclic Graphs](#)
- patches
 - creating / [Creating patches](#), [How to do it...](#), [There's more...](#)
 - creating, from branches / [Creating patches from branches](#), [How to do it...](#), [There's more...](#)
 - applying / [Applying patches](#), [How to do it...](#), [There's more...](#)
 - sending / [Sending patches](#), [How to do it...](#), [There's more...](#)
 - exchanging / [Exchanging patches](#)
- patch management interfaces
 - defining / [External tools – patch management interfaces](#)
- peer-to-peer workflow
 - about / [The peer-to-peer or forking workflow](#)
 - advantages / [The peer-to-peer or forking workflow](#)
 - disadvantages / [The peer-to-peer or forking workflow](#)
- perfect commit message
 - about / [The perfect commit message](#)
 - meaningful subject, writing / [Writing a meaningful subject](#)
 - bulleted details lines, adding / [Adding bulleted details lines, when needed](#)
 - useful information, adding / [Tie other useful information](#)
 - special messages, for releases / [Special messages for releases](#)
 - conclusions / [Conclusions](#)
- personal Git server
 - building up, with web interface / [Building up a personal Git server with web interface](#)
- plumbing / [Listing branches](#)
- plumbing command / [The plumbing and porcelain Git commands](#)
- plumbing commands
 - versus porcelain commands / [Listing ignored files](#)
- Pomodoro Technique / [Split up features and tasks](#)
- porcelain command / [The plumbing and porcelain Git commands](#)
- porcelain commands
 - versus plumbing commands / [Listing ignored files](#)
- porcelain option
 - about / [There's more...](#)

- Portable Object / [Other built-in attributes](#)
- post-commit hook
 - about / [Commit process hooks](#)
- post-receive hook
 - about / [The post-receive hook](#)
- post-update hook
 - about / [The post-update hook \(legacy mechanism\)](#)
- post hooks
 - about / [Automating Git with hooks](#)
- pre-commit hook
 - about / [Commit process hooks](#)
- pre-push hook
 - about / [Other client-side hooks](#)
- pre-rebase hook
 - about / [Other client-side hooks](#)
- pre-receive hook
 - about / [The pre-receive hook](#)
- preceding command
 - options / [How to do it...](#)
- predefined revision formats
 - about / [Predefined and user defined output formats](#)
- Preemptive commit comments blog post
 - reference link / [Write commit messages before starting to code](#)
- pre hooks
 - about / [Automating Git with hooks](#)
- prepare-commit-msg hook
 - about / [Using a branch description in the commit message, How to do it..., Commit process hooks](#)
 - used, for creating commit message / [How to do it..., There's more...](#)
- prepushHook branch
 - about / [How to do it...](#)
- Pretty Good Privacy (PGP)
 - about / [Other categories and uses of notes](#)
- problems
 - Wrong problem / [The art of the change review](#)

- Does not work / [The art of the change review](#)
 - Fails best practices / [The art of the change review](#)
- progressive-stability branches / [The graduation, or progressive-stability branches workflow](#)
- progressive-stability branches workflow
 - about / [The graduation, or progressive-stability branches workflow](#)
- project
 - starting / [Starting a project](#)
 - work, dividing into repositories / [Dividing work into repositories](#)
 - collaboration workflow, selecting / [Selecting the collaboration workflow](#)
 - generated files, tracking / [Choosing which files to keep under version control](#)
 - working / [Working on a project](#)
 - topic branch, working / [Working on a topic branch](#)
 - base, deciding / [Deciding what to base your work on](#)
 - changes, splitting into logically separate steps / [Splitting changes into logically separate steps](#)
 - good commit message, writing / [Writing a good commit message](#)
 - changes, preparing for submission / [Preparing changes for submission](#)
- public key authentication
 - about / [Public key authentication for SSH](#)
- public repository
 - pushing to / [Pushing to a public repository](#)
- published history
 - rewriting / [The perils of rewriting published history](#)
 - consequences, of upstream rewrite / [The consequences of upstream rewrite](#)
 - upstream history rewrite, recovering / [Recovering from an upstream history rewrite](#)
- pull
 - undoing / [Undoing a merge or a pull](#)
- pull.rebase configuration / [Rebase and merge setup](#)
- pull request

- generating / [Generating a pull request](#)
- push-to-update hook
 - about / [Push-to-update hook for pushing to nonbare repositories](#)
- pushing
 - about / [Pushing to a public repository](#)
- push modes
 - about / [Push modes and their use](#)
 - simple push mode / [The simple push mode – the default](#)
 - matching push mode / [The matching push mode for maintainers](#)
 - upstream push mode / [The upstream push mode for the centralized workflow](#)
 - current push mode / [The current push mode for the blessed repository workflow](#)

Q

- --quiet option / [How to do it...](#)
- -q, --quiet option / [There's more...](#)

R

- -rq option
 - about / [How it works...](#)
- rebase
 - about / [Introduction, The consequences of upstream rewrite](#)
 - continuing, git rebase -continue used / [Continuing a rebase with merge conflicts, How to do it, How it works](#)
 - used, for changing author of commits / [Changing the author of commits using a rebase, How to do it..., How it works...](#)
- rebase and merge setup, Git configuration
 - about / [Rebase and merge setup](#)
 - pull.rebase / [Rebase and merge setup](#)
 - branch.autosetuprebase / [Rebase and merge setup](#)
 - branch.<name>.rebase / [Rebase and merge setup](#)
- rebasing
 - commits, to another branch / [Rebasing commits to another branch, How to do it..., How it works](#)
 - selective commits / [Rebasing selective commits interactively,](#)

[How to do it, There's more...](#)

- recommendations
 - defining / [Other recommendations](#)
 - recovering possibility / [Don't panic, recovery is almost always possible](#)
 - published history, changing / [Don't change the published history](#)
 - releases, numbering / [Numbering and tagging releases](#)
 - releases, tagging / [Numbering and tagging releases](#)
 - automating / [Automate what is possible](#)
- references (refs)
 - about / [Branches and tags](#)
- reflog / [An introduction to version control and Git](#)
- reflog command
 - about / [Viewing past Git actions with git reflog](#)
 - used, for viewing Git actions / [Viewing past Git actions with git reflog](#), [How to do it...](#), [How it works...](#)
- reflog shortnames / [Reflog shortnames](#)
- refs namespaces
 - about / [How to do it...](#)
- refspec
 - about / [The refspec exemplified](#)
 - using / [The refspec exemplified](#), [Getting ready](#), [How to do it...](#), [How it works...](#)
 - format / [How it works...](#)
- [/ Refspec – remote to local branch mapping specification](#)
- refspec fetch
 - about / [How to do it...](#)
- refspecs / [Listing and examining remotes](#)
- regular repositories
 - converting, to bare repositories / [Converting a regular repository to a bare one](#)
- release branch, GitFlow / [The release branch](#)
- release branches workflow
 - about / [The release and trunk branches workflow](#)
- release candidates / [The release and trunk branches workflow](#)
- release note

- generating / [The contents of the releases, How to do it..., How it works...](#)
- remote
 - adding, to local repository / [Adding a remote to a local repository](#)
 - about / [The origin remote](#)
 - default branch, setting of / [Setting the default branch of remote](#)
- remote-tracking branches
 - about / [Branches and tags](#)
 - upstreaming / [Upstream of remote-tracking branches](#)
- remote branch
 - tracking / [Tracking branches](#)
- remote branches
 - about / [Branches with remotes, Getting ready, There's more...](#)
 - pruning / [Pruning remote branches, Getting ready, How to do it..., There's more...](#)
- remote changes
 - downloading / [Downloading remote changes, Checking for modifications and downloading them](#)
- remote Git repository
 - adding / [Adding a new remote](#)
- remote helpers
 - about / [Credential helpers and remote helpers, Remote helpers](#)
- remote repositories
 - managing / [Managing remote repositories](#)
- remote repository
 - local branch, pushing to / [Pushing a local branch to a remote repository](#)
- remotes
 - working with / [Working with remotes](#)
 - modification, uploading to / [Uploading modifications to remotes](#)
 - new branch, pushing to / [Pushing a new branch to the remote](#)
 - about / [Branches and tags](#)
 - listing / [Listing and examining remotes](#)
 - examining / [Listing and examining remotes](#)
 - information, updating of / [Updating information about remotes](#)

- renaming / [Renaming remotes](#)
- foreign SCM repositories, using as / [Using foreign SCM repositories as remotes](#)
- remote tracking branches
 - deleting / [Deleting remote-tracking branches](#)
- remote transport helpers
 - about / [Remote transport helpers](#)
- remote URLs
 - modifying / [Changing the remote URLs](#)
- removed file conflict
 - resolving / [Resolving a removed file conflict](#)
 - edited file, keeping / [Keeping the edited file](#)
- rename detection / [Blame – the line-wise history of a file](#)
- rename tracking / [Blame – the line-wise history of a file](#)
- replacements mechanism
 - using / [Using the replacements mechanism](#)
 - defining / [The replacements mechanism](#)
 - histories, joining with git replace / [Example – joining histories with git replace](#)
 - grafts, defining / [Historical note – grafts](#)
 - publishing / [Publishing and retrieving replacements](#)
 - retrieving / [Publishing and retrieving replacements](#)
- repo / [Setting up a new repository](#)
- repositories
 - working with / [Working with repositories](#)
 - file, unstaging / [Unstaging a file](#)
 - backing up / [Backup repositories](#)
 - backing up, as mirror repositories / [Back up your repositories as mirror repositories](#), [How to do it...](#), [How it works...](#), [There's more...](#), [A quick submodule how-to](#)
 - interacting with / [Interacting with other repositories](#)
 - comparing / [Environment variables affecting repository locations](#)
- repository
 - cloning / [Cloning a repository](#)
 - forking / [Forking a repository](#)

- archiving / [Archiving the repository](#)
 - bundling / [Bundling the repository](#)
 - notes, retrieving from / [Retrieving notes from the remote repository](#), [Getting ready](#), [How to do it...](#), [How it works...](#)
 - notes, pushing to / [Pushing notes to a remote repository](#), [How to do it...](#), [There's more...](#)
 - commit, tagging in / [Tagging commits in the repository](#), [Getting ready](#), [How to do it...](#), [There's more...](#)
 - splitting / [Splitting a repository](#), [Getting ready](#), [How to do it...](#), [How it works...](#)
 - updating, bundle used / [Using bundle to update an existing repository](#)
- repository history
 - rewriting / [How to do it...](#), [How it works...](#)
 - editing, with reposurgeon / [Editing the repository history with reposurgeon](#)
- repository information
 - displaying / [Displaying the repository information](#), [There's more...](#)
- repository maintenance
 - about / [Repository maintenance](#)
- repository only configurations / [Repository level](#)
- repository setup
 - about / [Repository setup](#)
 - Git repository, creating / [Creating a Git repository](#)
 - repository, cloning / [Cloning the repository and creating the first commit](#)
 - first commit, creating / [Cloning the repository and creating the first commit](#)
 - changes, publishing / [Publishing changes](#), [Examining history and viewing changes](#), [Renaming and moving files](#)
 - repository, updating / [Updating your repository \(with merge\)](#)
 - tag, creating / [Creating a tag](#)
- repository structure
 - about / [Repository structure and file status life cycle](#)
- reposurgeon

- repository history, editing with / [Editing the repository history with reposurgeon](#)
- repo tool
 - URL / [Third-party subproject management solutions](#)
- Request For Comments (RFC)
 - about / [Submitting and describing changes](#)
- reset command / [How it works...](#)
 - about / [Fixing mistakes with the reset command](#)
 - commits, squashing with / [Squashing commits with reset](#)
 - commit, splitting with / [Splitting a commit with reset](#)
- rev-parse command
 - examples / [Displaying the repository information](#)
 - about / [There's more...](#)
- reverse ancestry references / [Reverse ancestry references: the git describe output](#)
- revert command / [How it works...](#)
- review
 - responding to / [Responding to reviews and comments](#)
- reviewer preferences
 - about / [The art of the change review](#)
- revision
 - changes, saving in / [Searching changes in revisions](#)
 - viewing / [Viewing a revision and a file at revision](#)
- revision metadata
 - matching / [Matching revision metadata](#)
- revision range
 - selecting / [Selecting the revision range](#)
 - single revision, using as / [Single revision as a revision range](#)
 - for single revision / [The revision range for a single revision](#)
- revision range notation
 - using, in diff / [Triple-dot notation](#)
- revisions
 - including / [Multiple points – including and excluding revisions](#)
 - excluding / [Multiple points – including and excluding revisions](#)
- root nodes / [Directed Acyclic Graphs](#)

S

- --squash option / [How to do it...](#)
- --stdout option / [There's more...](#)
- --suffix=<sfx> option / [There's more...](#)
- -s, --signoff option / [There's more...](#)
- safer reset
 - about / [Safer reset – keeping your changes](#)
 - changes, rebasing to earlier revision / [Rebase changes to an earlier revision](#)
- SCM Manager
 - about / [The SCM Manager](#)
 - URL / [The SCM Manager](#)
- scripted rewrite
 - used, with git filter-branch / [Scripted rewrite with the git filter-branch](#)
- Secure Shell (SSH)
 - about / [SSH protocol](#)
- selective commit
 - about / [Selective commit](#)
 - files, sending to / [Selecting files to commit](#)
 - changes, selecting interactively / [Interactively selecting changes](#)
 - creating step by step, with staging area / [Creating a commit step by step](#)
- selective commits
 - rebasing / [Rebasing selective commits interactively](#), [Getting ready](#), [How to do it](#), [There's more...](#)
- semantic versioning
 - URL / [Numbering and tagging releases](#)
- send-email command / [There's more...](#)
- server-side hooks
 - about / [Automating Git with hooks](#), [Server-side hooks](#), [Server-side hooks](#)
 - pre-receive hook / [The pre-receive hook](#)
 - push-to-update hook / [Push-to-update hook for pushing to nonbare repositories](#)

- update hook / [The update hook](#)
 - post-receive hook / [The post-receive hook](#)
 - post-update hook / [The post-update hook \(legacy mechanism\)](#)
- SHA-1 hash function / [SHA-1 and the shortened SHA-1 identifier](#)
- shallow clone
 - about / [Historical note – grafts](#)
- shell prompt / [Git-aware command prompt](#)
- short-lived branches
 - about / [Short-lived branches](#)
 - feature branches / [Topic or feature branches](#)
 - topic branches / [Topic or feature branches](#)
 - bugfix branches / [Bugfix branches](#)
 - anonymous branches / [Detached HEAD – the anonymous branch](#)
- shortened SHA-1 identifier / [SHA-1 and the shortened SHA-1 identifier](#)
- shortlog git command / [How to do it...](#)
- signed commits
 - about / [Signed commits](#)
- signed pushes
 - about / [Signed pushes](#)
- signed tag / [Creating a tag](#)
 - verifying / [Tag verification](#)
- signed tags
 - about / [Branches and tags, Lightweight, annotated, and signed tags, Signed tags](#)
 - merging / [Merging signed tags \(merge tags\)](#)
- simple push mode
 - about / [The simple push mode – the default](#)
- single revision
 - using, as revision range / [Single revision as a revision range](#)
- single revision selection
 - about / [Single revision selection](#)
 - ancestry references / [Ancestry references](#)
 - by commit message / [Selecting revision by the commit message](#)
- smart HTTP(S) protocol

- about / [Smart HTTP\(S\) protocol](#), [Smart HTTP\(S\) protocol](#)
- smart transports
 - about / [Smart transports](#)
 - native Git protocol / [Native Git protocol](#)
 - SSH protocol / [SSH protocol](#)
 - smart HTTP(S) protocol / [Smart HTTP\(S\) protocol](#)
- smudge filter / [How to do it...](#)
- social coding
 - about / [Social coding – collaborate using GitHub](#)
 - repository, forking / [Forking a repository](#)
 - pull requests, submitting / [Submitting pull requests](#)
 - pull requests, creating / [Creating a pull request](#)
- source command
 - about / [How to do it...](#)
- source tree
 - bottlenecks, searching / [Finding bottlenecks in the source tree](#),
[How to do it...](#), [There's more...](#)
- SSH protocol
 - about / [SSH protocol](#)
- Stacked Git (StGit)
 - about / [External tools – patch management interfaces](#)
- staging area / [Getting ready](#)
 - about / [File statuses](#), [The staging area](#), [Stash and the staging area](#)
 - managing / [Managing worktrees and the staging area](#)
- standard layout
 - about / [Checking out the Subversion repository with svn client](#)
- stash
 - about / [Stash and the staging area](#)
 - un-applying / [Un-applying a stash](#)
- stash command / [How it works...](#), [There's more...](#)
- stashes
 - saving / [Saving and applying stashes](#), [How to do it...](#), [There's more...](#)
 - applying / [Saving and applying stashes](#), [How to do it...](#), [There's more...](#)

- recovering / [Recovering stashes that were dropped erroneously](#)
- stashing
 - about / [Stashing](#)
- stash internals
 - about / [Stash internals](#)
- state
 - saving, with WIP commit / [Saving and restoring state with the WIP commit](#)
 - restoring, with WIP commit / [Saving and restoring state with the WIP commit](#)
- status command / [There's more...](#)
- status information
 - bash prompt, used with / [Bash prompt with status information, How it works..., See also](#)
- StGit / [Amending a commit](#)
- subdirectory-filter
 - about / [There's more...](#)
- subfolder
 - transforming, into subtree or submodule / [Transforming a subfolder into a subtree or submodule](#)
- submodule
 - about / [A quick submodule how-to](#)
 - working / [Getting ready, How to do it..., There's more...](#)
 - repository, cloning / [There's more...](#)
 - versus subtree merging / [Submodule versus subtree merging](#)
- submodules
 - use cases / [Use cases for submodules](#)
- subtree
 - using, URL / [See also](#)
- subtree merging
 - about / [Subtree merging, How to do it..., How it works..., See also](#)
 - versus submodule / [Submodule versus subtree merging](#)
- subtrees
 - versus submodules / [Subtrees versus submodules](#)
 - use cases / [Use cases for subtrees](#)

- Subversion commands
 - versus Git commands / [Comparing Git and Subversion commands](#)
- Subversion repository
 - working on / [Working on a Subversion repository using Git](#)
 - checking out, with svn client / [Checking out the Subversion repository with svn client](#)
 - cloning, from Git / [Cloning a Subversion repository from Git](#)
 - tag, adding / [Adding a tag and a branch](#)
 - branch, adding / [Adding a tag and a branch](#)
 - Git, using with / [Using Git with a Subversion repository](#)
 - cloning / [Cloning the Subversion repository](#)
- Subversion repository, migrating
 - about / [Migrating a Subversion repository](#)
 - list of Subversion users, retrieving / [Retrieving the list of Subversion users](#)
 - ignored files, preserving / [Preserving the ignored file list](#)
 - pushing, to local bare Git repository / [Pushing to a local bare Git repository](#)
 - branches, adding / [Arranging branches and tags](#)
 - tags, adding / [Arranging branches and tags](#)
 - trunk branch, renaming to master / [Renaming the trunk branch to master](#)
 - local repository, pushing to remote / [Pushing the local repository to a remote](#)
- Subversion tags
 - converting, to Git tags / [Converting Subversion tags to Git tags](#)
- super project
 - recloning / [Getting ready, How to do it...](#)
- svn client
 - Subversion repository, checking out with / [Checking out the Subversion repository with svn client](#)
- symbolic references / [Symbolic references](#)
- system-wide configurations / [System level](#)
- SYSTEM layer
 - configuring / [Configuration targets, How to do it..., There's](#)

[more...](#)

T

- tab completion / [Command-line completion for Git](#)
- tag
 - about / [Branches and tags](#)
- tagging, commit
 - in repository / [Tagging commits in the repository](#), [Getting ready](#), [How to do it...](#), [There's more...](#)
- tag object
 - about / [The tag object](#), [Branches and tags](#), [Git objects](#)
- tag objects
 - about / [Annotated tags](#), [Git objects](#)
- tag operation
 - about / [Branches and tags](#)
- tag references
 - about / [Branch and tag references](#)
- tags
 - about / [Tagging commits in the repository](#)
 - lightweight tag / [Tagging commits in the repository](#)
 - annotated tag / [Tagging commits in the repository](#)
 - publishing / [Publishing tags](#)
- template
 - creating / [Templates](#), [How to do it...](#)
- template directory
 - creating / [A .git directory template](#), [Getting ready](#), [How to do it...](#), [How it works...](#)
- third-party subproject management solutions / [Third-party subproject management solutions](#)
- tilde character / [Ancestry references](#)
- time-limiting options
 - about / [Time-limiting options](#)
- time metaphor
 - about / [The time metaphor](#)
 - past / [The past](#)
 - present / [The present](#)

- future / [The future](#)
- tips and tricks, for hosting repositories
 - about / [Tips and tricks for hosting repositories](#)
 - size taken by repositories, reducing / [Reducing the size taken by repositories](#)
 - smart protocols, speeding up with pack bitmaps / [Speeding up smart protocols with pack bitmaps](#)
 - large nonresumable initial clone problem, solving / [Solving the large nonresumable initial clone problem](#)
- top contributor
 - extracting / [Extracting the top contributor, How to do it..., There's more...](#)
- topic branch
 - about / [Working on a topic branch](#)
- topic branches workflow
 - about / [The topic branches workflow](#)
 - graduation branches / [Graduation branches in a topic branch workflow](#)
 - branch management / [Branch management for a release in a topic branch workflow](#)
- TortoiseGit
 - about / [TortoiseGit](#)
- TortoiseSVN
 - URL / [Prerequisites](#)
 - about / [TortoiseGit](#)
- tracked files
 - about / [Ignoring files](#)
- tracking-related variables
 - GIT_TRACE / [Troubleshooting Git](#)
 - GIT_TRACE_PACKET / [Troubleshooting Git](#)
 - GIT_TRACE_SETUP / [Troubleshooting Git](#)
 - GIT_TRACE_PERFORMANCE / [Troubleshooting Git](#)
- transport protocols
 - about / [Transport protocols](#)
 - local transport / [Local transport](#)
 - smart transports / [Smart transports](#)

- offline transport with bundles / [Offline transport with bundles](#)
 - remote transport helpers / [Remote transport helpers](#)
- transport relay
 - with remote helpers / [Transport relay with remote helpers](#)
- tree
 - archives, creating from / [Creating archives from a tree, How to do it..., There's more...](#)
- tree-filter
 - about / [There's more...](#)
- tree information
 - displaying / [Displaying the tree information, How to do it...](#)
- tree object
 - about / [The tree object, Git objects](#)
 - writing, to database / [Writing a tree object to the database, How it works...](#)
- tree objects
 - about / [Git objects](#)
- triangular workflows
 - about / [Support for triangular workflows](#)
- tricks
 - about / [Tricks](#)
 - bare repositories / [Bare repositories](#)
- triple dot notation / [Triple-dot notation](#)
- troubleshooting, Git
 - about / [Data recovery and troubleshooting, Troubleshooting Git](#)
- trunk / [The release and trunk branches workflow](#)
- trunk branches workflow
 - about / [The release and trunk branches workflow](#)
- typos
 - about / [Typos autocorrection](#)

U

- UI
 - coloring, in prompt / [Color UI in the prompt, How to do it...](#)
- update-index command
 - using / [How it works...](#)

- update hook
 - about / [The update hook](#)
- upstream push mode
 - about / [The upstream push mode for the centralized workflow](#)
- user-defined revision formats
 - about / [Predefined and user defined output formats](#)
- user-wide configurations / [Global level](#)

V

- VCS
 - about / [Introduction](#)
- Version control system / [An introduction to version control and Git](#)
- Version control system (VCS)
 - about / [Directed Acyclic Graphs](#)
- Version Control System (VCS) / [Building the right commit](#)
- Vim (Vi Improved) / [Modify a committed file](#)
- visual diff tool
 - using / [Using a visual diff tool](#)
- visual manner
 - Git, learning in / [Learning Git in a visual manner](#)

W

- 3-way merge
 - performing / [Performing a 3-way merge](#)
- web interface
 - personal Git server, building with / [Building up a personal Git server with web interface](#)
- web interfaces
 - about / [Tools to manage Git repositories](#)
- whitespace / [Formatting and whitespace](#)
- whole-tree commits
 - about / [Whole-tree commits](#)
- Windows
 - about / [Windows](#)
- WIP commit
 - state, saving with / [Saving and restoring state with the WIP](#)

[commit](#)

- state, restoring with / [Saving and restoring state with the WIP commit](#)

- workflows

- adopting / [Adopting a workflow – a wise act](#)
- centralized workflows / [Centralized workflows, How they work](#)
- feature branch workflow / [Feature branch workflow](#)

- workflows, branches

- release ranches workflow / [The release and trunk branches workflow](#)
- trunk branches workflow / [The release and trunk branches workflow](#)
- graduation branches workflow / [The graduation, or progressive-stability branches workflow](#)
- progressive-stability branches workflow / [The graduation, or progressive-stability branches workflow](#)
- topic branches workflow / [The topic branches workflow](#)
- Git-flow / [Git-flow – a successful Git branching model](#)
- security issue, fixing / [Fixing a security issue](#)

- working area

- cleaning / [Cleaning the working area](#)

- working directory

- about / [The working directory](#)
- file statuses / [File statuses](#)
- reverting, to clean state / [Undo – working with a dirty area, How to do it...](#)

- worktree

- about / [Ignoring files](#)

- worktrees

- managing / [Managing worktrees and the staging area](#)

- world-wise techniques

- about / [World-wide techniques](#)
- last commit message, modifying / [Changing the last commit message](#)
- changes, tracing in file / [Tracing changes in a file](#)
- cherry picking activity / [Cherry picking](#)

X

- x4y5z6 commit / [The first parent, Cherry picking](#)

Z

- zero-status
 - about / [There's more...](#)

Table of Contents

Git: Mastering Version Control	34
Git: Mastering Version Control	35
Credits	36
Preface	37
What this learning path covers	37
What you need for this learning path	39
Who this learning path is for	40
Reader feedback	41
Customer support	42
Downloading the example code	42
Errata	43
Piracy	43
Questions	44
I. Module 1	45
1. Getting Started with Git	46
Installing Git	47
Running our first Git command	50
Setting up a new repository	51
Adding a file	52
Commit the added file	53
Modify a committed file	54
Summary	59
2. Git Fundamentals – Working Locally	60
Repository structure and file status life cycle	60
The working directory	60
File statuses	60
The staging area	61
Unstaging a file	62
The time metaphor	64
The past	64

The present	65
The future	69
Working with repositories	70
Unstaging a file	70
Viewing the history	73
Anatomy of a commit	73
The commit snapshot	74
The commit hash	74
Author, e-mail, and date	74
Commit messages	75
Committing a bunch of files	75
Ignoring some files and folders by default	75
Highlighting an important commit – Git tags	78
Taking another way – Git branching	79
Anatomy of branches	79
Looking at the current branches	80
Creating a new branch	80
Switching from branch to branch	81
Understanding what happens under the hood	82
A bird's eye view to branches	85
Typing is boring – Git aliases	87
Merging branches	88
Merge is not the end of the branch	90
Exercises	90
Exercise 2.1	90
What you will learn	90
Scenario	90
Results	91
Exercise 2.2	91
What you will learn	91
Scenario	91
Results	91

Deal with branches' modifications	91
Diffing branches	92
Using a visual diff tool	95
Resolving merge conflicts	96
Edit collisions	96
Resolving a removed file conflict	98
Keeping the edited file	98
Resolving conflicts by removing the file	100
Stashing	102
Summary	105
3. Git Fundamentals – Working Remotely	106
Working with remotes	106
Setting up a new GitHub account	107
Cloning a repository	110
Uploading modifications to remotes	111
What do I send to the remote when I push?	112
Pushing a new branch to the remote	113
The origin	114
Tracking branches	115
Downloading remote changes	116
Checking for modifications and downloading them	116
Applying downloaded changes	118
Going backward: publish a local repository to GitHub	119
Adding a remote to a local repository	121
Pushing a local branch to a remote repository	121
Social coding – collaborate using GitHub	122
Forking a repository	122
Submitting pull requests	125
Creating a pull request	126
Summary	131
4. Git Fundamentals – Niche Concepts, Configurations, and Commands	132

Dissecting the Git configuration	132
Configuration architecture	132
Configuration levels	132
System level	133
Global level	134
Repository level	134
Listing configurations	134
Editing configuration files manually	135
Setting up other environment configurations	135
Basic configurations	135
Typos autocorrection	136
Push default	137
Defining the default editor	138
Other configurations	138
Git aliases	139
Shortcuts to common commands	139
Creating commands	139
git unstage	140
git undo	140
git last	140
git difflast	141
Advanced aliases with external commands	141
Removing an alias	142
Aliasing the git command itself	142
Git references	143
Symbolic references	143
Ancestry references	144
The first parent	144
The second parent	145
World-wide techniques	147
Changing the last commit message	147

Tracing changes in a file	147
Cherry picking	149
Tricks	152
Bare repositories	152
Converting a regular repository to a bare one	152
Backup repositories	153
Archiving the repository	153
Bundling the repository	153
Summary	155
5. Obtaining the Most – Good Commits and Workflows	156
The art of committing	156
Building the right commit	156
Make only one change per commit	157
Split up features and tasks	158
Write commit messages before starting to code	160
Include the whole change in one commit	161
Describe the change, not what you have done	161
Don't be afraid to commit	162
Isolate meaningless commits	163
The perfect commit message	164
Writing a meaningful subject	164
Adding bulleted details lines, when needed	165
Tie other useful information	165
Special messages for releases	165
Conclusions	166
Adopting a workflow – a wise act	167
Centralized workflows	167
How they work	168
Feature branch workflow	168
GitFlow	169
The master branch	170

Hotfixes branches	171
The develop branch	171
The release branch	172
The feature branches	173
Conclusion	173
The GitHub flow	173
Anything in the master branch is deployable	174
Creating descriptive branches off of the master	175
Pushing to named branches constantly	175
Opening a pull request at any time	175
Merging only after a pull request review	176
Deploying immediately after review	176
Conclusions	176
Other workflows	177
The Linux kernel workflow	177
Summary	179
6. Migrating to Git	180
Before starting	180
Prerequisites	180
Working on a Subversion repository using Git	181
Creating a local Subversion repository	181
Checking out the Subversion repository with svn client	181
Cloning a Subversion repository from Git	183
Setting up a local Subversion server	183
Adding a tag and a branch	185
Committing a file to Subversion using Git as a client	185
Using Git with a Subversion repository	187
Migrating a Subversion repository	188
Retrieving the list of Subversion users	188
Cloning the Subversion repository	188
Preserving the ignored file list	189

Pushing to a local bare Git repository	189
Arranging branches and tags	190
Renaming the trunk branch to master	190
Converting Subversion tags to Git tags	190
Pushing the local repository to a remote	191
Comparing Git and Subversion commands	192
Summary	194
7. Git Resources	195
Git GUI clients	195
Windows	195
Git GUI	195
TortoiseGit	196
GitHub for Windows	197
Atlassian SourceTree	198
Cmder	199
Mac OS X	200
Linux	201
Building up a personal Git server with web interface	202
The SCM Manager	202
Learning Git in a visual manner	204
Git on the Internet	205
Git community on Google+	205
GitMinutes and Thomas Ferris Nicolaisen's blog	205
Ferdinando Santacroce's blog	205
Summary	206
II. Module 2	207
1. Navigating Git	208
Introduction	208
Git's objects	211
Getting ready	211
How to do it...	211

The commit object	212
The tree object	212
The blob object	213
The branch	214
The tag object	215
How it works...	215
There's more...	216
See also	216
The three stages	217
Getting ready	217
How to do it...	217
How it works...	219
See also	221
Viewing the DAG	222
Getting ready	222
How to do it...	223
How it works...	225
See also	225
Extracting fixed issues	226
Getting ready	226
How to do it...	226
How it works...	228
There's more...	228
Getting a list of the changed files	230
Getting ready	230
How to do it...	230
How it works...	230
There's more...	230
See also	231
Viewing history with Gitk	232
Getting ready	232

How to do it...	232
How it works...	233
There's more...	233
Finding commits in history	235
Getting ready	235
How to do it...	235
How it works...	235
There's more...	236
Searching through history code	238
Getting ready	238
How to do it...	238
How it works...	238
There's more...	239
2. Configuration	241
Configuration targets	241
Getting ready	241
How to do it...	241
How it works...	243
There's more...	243
Querying the existing configuration	245
Getting ready	245
How to do it...	245
How it works...	246
There's more...	246
Templates	248
Getting ready	248
How to do it...	248
How it works...	249
A .git directory template	250
Getting ready	250
How to do it...	251
How it works...	252

See also	253
A few configuration examples	254
Getting ready	254
How to do it...	254
Rebase and merge setup	254
Expiry of objects	255
Autocorrect	256
How it works...	257
There's more...	257
Git aliases	258
Getting ready	258
How to do it...	258
How it works...	262
There's more...	262
The refspec exemplified	263
Getting ready	263
How to do it...	264
How it works...	266
3. Branching, Merging, and Options	267
Introduction	267
Managing your local branches	268
Getting ready	268
How to do it...	268
How it works...	269
There's more...	269
Branches with remotes	272
Getting ready	272
How to do it...	273
There's more...	274
Forcing a merge commit	276
Getting ready	276
How to do it...	276

There's more...	278
Using git rerere to merge known conflicts	281
How to do it...	281
There's more...	285
The difference between branches	288
Getting ready	288
How to do it...	288
There's more...	289
4. Rebase Regularly and Interactively, and Other Use Cases	290
Introduction	290
Rebasing commits to another branch	291
Getting ready	291
How to do it...	291
How it works	292
Continuing a rebase with merge conflicts	293
How to do it	293
How it works	294
There's more...	295
Rebasing selective commits interactively	296
Getting ready	296
How to do it	296
There's more...	298
Squashing commits using an interactive rebase	300
Getting ready	300
How to do it...	300
There's more...	303
Changing the author of commits using a rebase	307
Getting ready	307
How to do it...	307
How it works...	310
Auto-squashing commits	311
Getting ready	311

How to do it...	311
There's more...	315
5. Storing Additional Information in Your Repository	316
Introduction	316
Adding your first Git note	317
Getting ready	317
How to do it...	317
There's more...	320
Separating notes by category	322
Getting ready	322
How to do it...	322
How it works...	325
Retrieving notes from the remote repository	327
Getting ready	327
How to do it...	328
How it works...	330
Pushing notes to a remote repository	331
How to do it...	331
There's more...	333
Tagging commits in the repository	334
Getting ready	334
How to do it...	335
There's more...	336
6. Extracting Data from the Repository	340
Introduction	340
Extracting the top contributor	341
Getting ready	341
How to do it...	341
There's more...	344
Finding bottlenecks in the source tree	346
Getting ready	346
How to do it...	346

There's more...	350
Grepping the commit messages	353
Getting ready	353
How to do it...	353
The contents of the releases	356
How to do it...	356
How it works...	357
7. Enhancing Your Daily Work with Git Hooks, Aliases, and Scripts	359
Introduction	359
Using a branch description in the commit message	360
Getting ready	360
How to do it...	361
Creating a dynamic commit message template	365
Getting ready	365
How to do it...	365
There's more...	368
Using external information in the commit message	372
Getting ready	372
How to do it...	372
Preventing the push of specific commits	377
Getting ready	377
How to do it...	377
There's more...	380
Configuring and using Git aliases	383
How to do it...	383
How it works...	387
Configuring and using Git scripts	389
How to do it...	389
Setting up and using a commit template	391
Getting ready	391
How to do it...	391

8. Recovering from Mistakes	394
Introduction	394
Undo – remove a commit completely	396
Getting ready	396
How to do it...	396
How it works...	397
Undo – remove a commit and retain the changes to files	399
Getting ready	399
How to do it...	399
How it works...	400
Undo – remove a commit and retain the changes in the staging area	402
Getting ready	402
How to do it...	402
How it works...	403
Undo – working with a dirty area	405
Getting ready	405
How to do it...	406
How it works...	408
See also	408
Redo – recreate the latest commit with new changes	409
Getting ready	409
How to do it...	409
How it works...	412
There is more...	412
Revert – undo the changes introduced by a commit	414
Getting ready	414
How to do it...	414
How it works...	415
There's more...	415
Reverting a merge	417

Getting ready	417
How to do it...	418
How it works...	419
There is more...	420
See also	421
Viewing past Git actions with git reflog	423
Getting ready	423
How to do it...	423
How it works...	425
Finding lost changes with git fsck	426
Getting ready	426
How to do it...	426
How it works...	428
See also	429
9. Repository Maintenance	430
Introduction	430
Pruning remote branches	431
Getting ready	431
How to do it...	432
How it works...	433
There's more...	433
Running garbage collection manually	434
Getting ready	434
How to do it...	435
How it works...	437
Turning off automatic garbage collection	438
Getting ready	438
How to do it...	438
Splitting a repository	440
Getting ready	440
How to do it...	442

How it works...	444
There's more...	445
Rewriting history – changing a single file	446
Getting ready	446
How to do it...	446
How it works...	447
Back up your repositories as mirror repositories	448
Getting ready	448
How to do it...	448
How it works...	450
There's more...	450
A quick submodule how-to	452
Getting ready	452
How to do it...	452
There's more...	456
Subtree merging	458
Getting ready	458
How to do it...	458
How it works...	464
See also	465
Submodule versus subtree merging	466
10. Patching and Offline Sharing	467
Introduction	467
Creating patches	468
Getting ready	468
How to do it...	468
How it works...	471
There's more...	471
Creating patches from branches	472
Getting ready	472
How to do it...	472

How it works...	473
There's more...	473
Applying patches	474
Getting ready	474
How to do it...	474
How it works...	475
There's more...	475
Sending patches	477
Getting ready	477
How to do it...	477
How it works...	480
There's more...	480
Creating Git bundles	482
Getting ready	482
How to do it...	482
How it works...	484
Using a Git bundle	485
Getting ready	485
How to do it...	485
There's more...	488
Creating archives from a tree	489
Getting ready	489
How to do it...	489
There's more...	490
11. Git Plumbing and Attributes	493
Introduction	493
Displaying the repository information	495
Getting ready	495
How to do it...	495
There's more...	496
Displaying the tree information	497
Getting ready	497

How to do it...	497
Displaying the file information	501
Getting ready	501
How to do it...	501
There's more...	503
Writing a blob object to the database	505
Getting ready	505
How to do it...	505
How it works...	506
There's more...	507
Writing a tree object to the database	508
Getting ready	508
How to do it...	508
How it works...	509
Writing a commit object to the database	510
Getting ready	510
How to do it...	510
How it works...	510
Keyword expansion with attribute filters	513
Getting ready	513
How to do it...	513
How it works...	514
There's more...	515
Metadata diff of binary files	517
Getting ready	517
How to do it...	517
How it works...	518
There's more...	519
Storing binaries elsewhere	520
Getting ready	520
How to do it...	520
How it works...	523

There's more...	524
See also	525
Checking the attributes of a file	526
Getting ready	526
How to do it...	526
Attributes to export an archive	528
Getting ready	528
How to do it...	528
There's more...	529
12. Tips and Tricks	531
Introduction	531
Using git stash	532
Getting ready	532
How to do it...	533
How it works...	535
There's more...	536
Saving and applying stashes	539
Getting ready	539
How to do it...	539
There's more...	540
Debugging with git bisect	542
Getting ready	542
How to do it...	543
There's more...	546
Using the blame command	548
Getting ready	548
How to do it...	548
There's more...	549
Color UI in the prompt	550
Getting ready	550
How to do it...	550
There's more...	551

Autocompletion	553
Getting ready	553
Linux	553
Mac	553
Windows	553
How to do it...	554
How it works...	554
There's more...	554
Bash prompt with status information	555
Getting ready	555
How to do it...	555
How it works...	555
There's more...	557
See also	558
More aliases	559
Getting ready	559
How to do it...	559
Interactive add	564
Getting ready	564
How to do it...	564
There's more...	568
Interactive add with Git GUI	570
Getting ready	570
How to do it...	570
Ignoring files	574
Getting ready	574
How to do it...	574
There's more...	576
See also...	577
Showing and cleaning ignored files	578
Getting ready	578
How to do it...	578

There's more...	579
III. Module 3	580
1. Git Basics in Practice	581
An introduction to version control and Git	581
Git by example	583
Repository setup	583
Creating a Git repository	584
Cloning the repository and creating the first commit	585
Publishing changes	587
Examining history and viewing changes	588
Renaming and moving files	591
Updating your repository (with merge)	592
Creating a tag	594
Resolving a merge conflict	595
Adding files in bulk and removing files	598
Undoing changes to a file	599
Creating a new branch	600
Merging a branch (no conflicts)	601
Undoing an unpublished merge	602
Summary	604
2. Exploring Project History	605
Directed Acyclic Graphs	605
Whole-tree commits	608
Branches and tags	608
Branch points	611
Merge commits	611
Single revision selection	613
HEAD – the implicit revision	613
Branch and tag references	614
SHA-1 and the shortened SHA-1 identifier	615
Ancestry references	617

Reverse ancestry references: the git describe output	618
Reflog shortnames	618
Upstream of remote-tracking branches	619
Selecting revision by the commit message	620
Selecting the revision range	622
Single revision as a revision range	622
Double dot notation	622
Multiple points – including and excluding revisions	625
The revision range for a single revision	626
Triple-dot notation	626
Searching history	629
Limiting the number of revisions	629
Matching revision metadata	629
Time-limiting options	629
Matching commit contents	630
Commit parents	631
Searching changes in revisions	632
Selecting types of change	634
History of a file	635
Path limiting	635
History simplification	636
Blame – the line-wise history of a file	637
Finding bugs with git bisect	640
Selecting and formatting the git log output	643
Predefined and user defined output formats	643
Including, formatting, and summing up changes	645
Summarizing contributions	647
Viewing a revision and a file at revision	649
Summary	651
3. Developing with Git	652
Creating a new commit	652

The DAG view of creating a new commit	653
The index – a staging area for commits	654
Examining the changes to be committed	657
The status of the working directory	657
Examining differences from the last revision	660
Unified Git diff format	662
Selective commit	667
Selecting files to commit	668
Interactively selecting changes	668
Creating a commit step by step	670
Amending a commit	671
Working with branches	674
Creating a new branch	675
Creating orphan branches	676
Selecting and switching to a branch	677
Obstacles to switching to a branch	677
Anonymous branches	678
Git checkout DWIM-mery	679
Listing branches	680
Rewinding or resetting a branch	681
Deleting a branch	683
Changing the branch name	684
Summary	686
4. Managing Your Worktree	687
Ignoring files	687
Marking files as intentionally untracked	689
Which types of file should be ignored?	691
Listing ignored files	693
Ignoring changes in tracked files	694
File attributes	697
Identifying binary files and end-of-line conversions	698

Diff and merge configuration	700
Generating diffs and binary files	700
Configuring diff output	702
Performing a 3-way merge	703
Transforming files (content filtering)	704
Obligatory file transformations	706
Keyword expansion and substitution	707
Other built-in attributes	709
Defining attribute macros	710
Fixing mistakes with the reset command	711
Rewinding the branch head, softly	711
Removing or amending a commit	712
Squashing commits with reset	713
Resetting the branch head and the index	713
 Splitting a commit with reset	714
Saving and restoring state with the WIP commit	714
Discarding changes and rewinding branch	715
Moving commits to a feature branch	716
Undoing a merge or a pull	717
Safer reset – keeping your changes	717
Rebase changes to an earlier revision	718
Stashing away your changes	719
Using git stash	719
Stash and the staging area	720
Stash internals	721
Un-applying a stash	723
Recovering stashes that were dropped erroneously	723
Managing worktrees and the staging area	725
Examining files and directories	725
Searching file contents	726
Un-tracking, un-staging, and un-modifying files	727

Resetting a file to the old version	729
Cleaning the working area	730
Multiple working directories	732
Summary	733
5. Collaborative Development with Git	735
Collaborative workflows	735
Bare repositories	736
Interacting with other repositories	737
The centralized workflow	738
The peer-to-peer or forking workflow	740
The maintainer or integration manager workflow	741
The hierarchical or dictator and lieutenants workflows	743
Managing remote repositories	746
The origin remote	746
Listing and examining remotes	747
Adding a new remote	748
Updating information about remotes	749
Renaming remotes	749
Changing the remote URLs	749
Changing the list of branches tracked by remote	750
Setting the default branch of remote	750
Deleting remote-tracking branches	751
Support for triangular workflows	751
Transport protocols	753
Local transport	753
Smart transports	755
Native Git protocol	755
SSH protocol	756
Smart HTTP(S) protocol	757
Offline transport with bundles	757
Cloning and updating with bundle	759

Using bundle to update an existing repository	762
Utilizing bundle to help with the initial clone	763
Remote transport helpers	764
Transport relay with remote helpers	765
Using foreign SCM repositories as remotes	766
Credentials/password management	768
Asking for passwords	768
Public key authentication for SSH	769
Credential helpers	770
Publishing your changes upstream	773
Pushing to a public repository	773
Generating a pull request	774
Exchanging patches	775
Chain of trust	778
Content-addressed storage	778
Lightweight, annotated, and signed tags	779
Lightweight tags	779
Annotated tags	779
Signed tags	780
Publishing tags	780
Tag verification	780
Signed commits	781
Merging signed tags (merge tags)	782
Summary	785
6. Advanced Branching Techniques	787
Types and purposes of branches	787
Long-running, perpetual branches	789
Integration, graduation, or progressive-stability branches	789
Per-release branches and per-release maintenance	791
Hotfix branches for security fixes	792
Per-customer or per-deployment branches	792

Automation branches	792
Mob branches for anonymous push access	793
The orphan branch trick	793
Short-lived branches	795
Topic or feature branches	795
Bugfix branches	796
Detached HEAD – the anonymous branch	796
Branching workflows and release engineering	798
The release and trunk branches workflow	798
The graduation, or progressive-stability branches workflow	799
The topic branches workflow	801
Graduation branches in a topic branch workflow	803
Branch management for a release in a topic branch workflow	806
Git-flow – a successful Git branching model	808
Fixing a security issue	809
Interacting with branches in remote repositories	812
Upstream and downstream	812
Remote-tracking branches and refspec	813
Remote-tracking branches	813
Refspec – remote to local branch mapping specification	815
Fetching and pulling versus pushing	816
Pull – fetch and update current branch	816
Pushing to the current branch in a nonbare remote repository	817
The default fetch refspec and push modes	817
Fetching and pushing branches and tags	819
Fetching branches	819
Fetching tags and automatic tag following	820
Pushing branches and tags	821
Push modes and their use	821
The simple push mode – the default	822
The matching push mode for maintainers	822

The upstream push mode for the centralized workflow	824
The current push mode for the blessed repository workflow	825
Summary	827
7. Merging Changes Together	829
Methods of combining changes	829
Merging branches	830
No divergence – fast-forward and up-to-date cases	830
Creating a merge commit	833
Merge strategies and their options	835
Reminder – merge drivers	836
Reminder – signing merges and merging tags	836
Copying and applying a changeset	837
Cherry-pick – creating a copy of a changeset	837
Revert – undoing an effect of a commit	838
Applying a series of commits from patches	839
Cherry-picking and reverting a merge	840
Rebasing a branch	840
Merge versus rebase	842
Types of rebase	843
Advanced rebasing techniques	844
Resolving merge conflicts	847
The three-way merge	847
Examining failed merges	849
Conflict markers in the worktree	849
Three stages in the index	851
Examining differences – the combined diff format	851
How do we get there: git log --merge	853
Avoiding merge conflicts	854
Useful merge options	854
Rerere – reuse recorded resolutions	855
Dealing with merge conflicts	856
Aborting a merge	856

Selecting ours or theirs version	856
Scriptable fixes – manual file remerging	857
Using graphical merge tools	857
Marking files as resolved and finalizing merges	858
Resolving rebase conflicts	858
git-imerge – incremental merge and rebase for git	859
Summary	860
8. Keeping History Clean	861
An introduction to Git internals	862
Git objects	862
The plumbing and porcelain Git commands	865
Environment variables used by Git	866
Environment variables affecting global behavior	867
Environment variables affecting repository locations	868
Environment variables affecting committing	870
Rewriting history	871
Amending the last commit	871
An interactive rebase	872
Reordering, removing, and fixing commits	873
Squashing commits	877
Splitting commits	879
Testing each rebased commit	880
External tools – patch management interfaces	881
Scripted rewrite with the git filter-branch	882
Running the filter-branch without filters	883
Available filter types for filter-branch and their use	884
Examples of using the git filter-branch	887
External tools for large-scale history rewriting	891
Removing files from the history with BFG Repo Cleaner	891
Editing the repository history with reposurgeon	891
The perils of rewriting published history	892
The consequences of upstream rewrite	893

Recovering from an upstream history rewrite	895
Amending history without rewriting	898
Reverting a commit	898
Reverting a faulty merge	900
Recovering from reverted merges	902
Storing additional information with notes	906
Adding notes to a commit	906
How notes are stored	908
Other categories and uses of notes	909
Rewriting history and notes	913
Publishing and retrieving notes	914
Using the replacements mechanism	916
The replacements mechanism	916
Example – joining histories with git replace	917
Historical note – grafts	920
Publishing and retrieving replacements	920
Summary	922
9. Managing Subprojects – Building a Living Framework	923
Managing library and framework dependencies	924
Managing dependencies outside Git	926
Manually importing the code into your project	927
A Git subtree for embedding the subproject code	929
Creating a remote for a subproject	932
Adding a subproject as a subtree	932
Cloning and updating superprojects with subtrees	935
Getting updates from subprojects with a subtree merge	935
Showing changes between a subtree and its upstream	938
Sending changes to the upstream of a subtree	938
The Git submodules solution: repository inside repository	940
Gitlinks, .git files, and the git submodule command	941
Adding a subproject as a submodule	944
Cloning superprojects with submodules	947

Updating submodules after superproject changes	949
Examining changes in a submodule	951
Getting updates from the upstream of the submodule	953
Sending submodule changes upstream	954
Transforming a subfolder into a subtree or submodule	955
Subtrees versus submodules	957
Use cases for subtrees	958
Use cases for submodules	959
Third-party subproject management solutions	960
Managing large Git repositories	962
Handling repositories with a very long history	962
Using shallow clones to get truncated history	963
Cloning only a single branch	964
Handling repositories with large binary files	964
Splitting the binary asset folder into a separate submodule	965
Storing large binary files outside the repository	965
Summary	968
10. Customizing and Extending Git	969
Git on the command line	970
Git-aware command prompt	970
Command-line completion for Git	974
Autocorrection for Git commands	975
Making the command line prettier	976
Alternative command line	977
Graphical interfaces	979
Types of graphical tools	979
Graphical diff and merge tools	980
Graphical interface examples	982
Configuring Git	984
Command-line options and environment variables	984
Git configuration files	985
The syntax of Git configuration files	986

Accessing the Git configuration	988
Basic client-side configuration	990
The rebase and merge setup, configuring pull	992
Preserving undo information – the expiry of objects	993
Formatting and whitespace	993
Server-side configuration	996
Per-file configuration with gitattributes	997
Automating Git with hooks	1000
Installing a Git hook	1000
A template for repositories	1001
Client-side hooks	1002
Commit process hooks	1002
Hooks for applying patches from e-mails	1005
Other client-side hooks	1006
Server-side hooks	1008
Extending Git	1011
Command aliases for Git	1011
Adding new Git commands	1013
Credential helpers and remote helpers	1014
Summary	1016
11. Git Administration	1017
Repository maintenance	1018
Data recovery and troubleshooting	1020
Recovering a lost commit	1020
Troubleshooting Git	1022
Git on the server	1024
Server-side hooks	1024
The pre-receive hook	1025
Push-to-update hook for pushing to nonbare repositories	1026
The update hook	1027
The post-receive hook	1028

The post-update hook (legacy mechanism)	1029
Using hooks to implement the Git-enforced policy	1029
Enforcing the policy with server-side hooks	1029
Early notices about policy violations with client-side hooks	1031
Signed pushes	1032
Serving Git repositories	1033
Local protocol	1034
SSH protocol	1034
Anonymous Git protocol	1035
Smart HTTP(S) protocol	1037
Dumb protocols	1038
Remote helpers	1039
Tools to manage Git repositories	1040
Tips and tricks for hosting repositories	1041
Reducing the size taken by repositories	1041
Speeding up smart protocols with pack bitmaps	1042
Solving the large nonresumable initial clone problem	1043
Augmenting development workflows	1044
Summary	1045
12. Git Best Practices	1046
Starting a project	1046
Dividing work into repositories	1046
Selecting the collaboration workflow	1047
Choosing which files to keep under version control	1047
Working on a project	1049
Working on a topic branch	1049
Deciding what to base your work on	1050
Splitting changes into logically separate steps	1051
Writing a good commit message	1052
Preparing changes for submission	1054
Integrating changes	1056

Submitting and describing changes	1056
The art of the change review	1058
Responding to reviews and comments	1060
Other recommendations	1062
Don't panic, recovery is almost always possible	1062
Don't change the published history	1062
Numbering and tagging releases	1063
Automate what is possible	1064
Summary	1065
A. Bibliography	1066
Index	1067