THE EXPERT'S VOICE®

SECOND EDITION

Pro Git

EVERYTHING YOU NEED TO KNOW ABOUT GIT

Scott Chacon and Ben Straub

Apress®

Pro Git

Scott Chacon, Ben Straub

Version 2.1.448, 2025-07-25

Table of Contents

License

Preface by Scott Chacon

Welcome to the second edition of Pro Git. The first edition was published over four years ago now. Since then a lot has changed and yet many important things have not. While most of the core commands and concepts are still valid today as the Git core team is pretty fantastic at keeping things backward compatible, there have been some significant additions and changes in the community surrounding Git. The second edition of this book is meant to address those changes and update the book so it can be more helpful to the new user.

When I wrote the first edition, Git was still a relatively difficult to use and barely adopted tool for the harder core hacker. It was starting to gain steam in certain communities, but had not reached anywhere near the ubiquity it has today. Since then, nearly every open source community has adopted it. Git has made incredible progress on Windows, in the explosion of graphical user interfaces to it for all platforms, in IDE support and in business use. The Pro Git of four years ago knows about none of that. One of the main aims of this new edition is to touch on all of those new frontiers in the Git community.

The Open Source community using Git has also exploded. When I originally sat down to write the book nearly five years ago (it took me a while to get the first version out), I had just started working at a very little known company developing a Git hosting website called GitHub. At the time of publishing there were maybe a few thousand people using the site and just four of us working on it. As I write this introduction, GitHub is announcing our 10 millionth hosted project, with nearly 5 million registered developer accounts and over 230 employees. Love it or hate it, GitHub has heavily changed large swaths of the Open Source community in a way that was barely conceivable when I sat down to write the first edition.

I wrote a small section in the original version of Pro Git about GitHub as an example of hosted Git which I was never very comfortable with. I didn't much like that I was writing what I felt was essentially a community resource and also talking about my company in it. While I still don't love that conflict of interests, the importance of GitHub in the Git community is unavoidable. Instead of an example of Git hosting, I have decided to turn that part of the book into more deeply describing what GitHub is and how to effectively use it. If you are going to learn how to use Git then knowing how to use GitHub will help you take part in a huge community, which is valuable no matter which Git host you decide to use for your own code.

The other large change in the time since the last publishing has been the development and rise of the HTTP protocol for Git network transactions. Most of the examples in the book have been changed to HTTP from SSH because it's so much simpler.

It's been amazing to watch Git grow over the past few years from a relatively obscure version control system to basically dominating commercial and open source version control. I'm happy that Pro Git has done so well and has also been able to be one of the few technical books on the market that is both quite successful and fully open source.

I hope you enjoy this updated edition of Pro Git.

Preface by Ben Straub

The first edition of this book is what got me hooked on Git. This was my introduction to a style of making software that felt more natural than anything I had seen before. I had been a developer for several years by then, but this was the right turn that sent me down a much more interesting path than the one I was on.

Now, years later, I'm a contributor to a major Git implementation, I've worked for the largest Git hosting company, and I've traveled the world teaching people about Git. When Scott asked if I'd be interested in working on the second edition, I didn't even have to think.

It's been a great pleasure and privilege to work on this book. I hope it helps you as much as it did me.

Dedications

To my wife, Becky, without whom this adventure never would have begun. — Ben

This edition is dedicated to my girls. To my wife Jessica who has supported me for all of these years and to my daughter Josephine, who will support me when I'm too old to know what's going on. — Scott

Contributors

Since this is an Open Source book, we have gotten several errata and content changes donated over the years. Here are all the people who have contributed to the English version of Pro Git as an open source project. Thank you everyone for helping make this a better book for everyone.

```
Contributors as of ece0b7f5:

4wk-                            Johannes Schindelin             Sean Head
Adam Laflamme                   John Lin                        Sean Jacobs
Adrien Ollier                   Jon Forrest                     Sebastian Krause
Akrom K                         Jon Freed                       Sergey Kuznetsov
Alan D. Salewski                Jonathan                        Severino Lorilla Jr
Alba Mendez                     Jordan Hayashi                  Shengbin Meng
Aleh Suprunovich                Joris Valette                   Sherry Hietala
Alex Povel                      Josh Byster                     Shi Yan
Alexander Bezzubov              Joshua Webb                     Siarhei Bobryk
Alexandre Garnier               Junjie Yuan                     Siarhei Krukau
Alfred Myers                    Junyeong Yim                    Skyper
Amanda Dillon                   Justin Clift                    Smaug123
Andreas Bjørnestad              Jörn Auerbach                   Snehal Shekatkar
Andrei Dascalu                  Kaartic Sivaraam                Solt Budavári
Andrei Korshikov                KatDwo                          Song Li
Andrew Blommestyn               Katrin Leinweber                Stephan van Maris
Andrew Kreimer                  Kausar Mehmood                  Steven Roddis
Andrew Layman                   Keith Hill                      Stuart P. Bentley
Andrew MacFie                   Kenneth Kin Lum                 SudarsanGP
Andrew Metcalf                  Kenneth Lum                     Suhaib Mujahid
Andrew Murphy                   Klaus Frank                     Susan Stevens
AndyGee                         Kristijan "Fremen" Velkovski    Sven Selberg
AnneTheAgile                    Krzysztof Szumny                Thanix
Anthony Loiseau                 Kyrylo Yatsenko                 Thomas Ackermann
Anton Trunov                    Károly Ozsvárt                  Thomas Hartmann
Antonello Piemonte              Lars Vogel                      Tiffany
Antonino Ingargiola             Laxman                          Tom Schady
Ardavast Dayleryan              Lazar95                         Tomas Fiers
Artem Leshchev                  Leonard Laszlo                  Tomoki Aonuma
Atul Varma                      Linus Heckemann                 Trevor Jobling
Bagas Sanjaya                   Logan Hasson                    Tvirus
Ben Sima                        Louise Corrigan                 Tyler Cipriani
Benjamin Dopplinger             Luc Morin                       Ud Yzr
Billy Griffin                   Lukas Röllin                    UgmaDevelopment
Bob Kline                       Marat Radchenko                 Vadim Markovtsev
Bohdan Pylypenko                Marcin Sędłak-Jakubowski        Vangelis Katsikaros
Borek Bernard                   Marie-Helene Burle              Vegar Vikan
Brett Cannon                    Marius Žilėnas                  Victor Ma
Buzut                           Markus KARG                     Vipul Kumar
C Nguyen                        Marti Bolivar                   Vitaly Kuznetsov
Cadel Watson                    Mashrur Mia (Sa'ad)             Volker Weißmann
```

Carlos Martín Nieto
Carlos Tafur
Chaitanya Gurrapu
Changwoo Park
Christian Decker
Christoph Bachhuber
Christoph Prokop
Christopher Wilson
CodingSpiderFox
Cory Donnelly
Cullen Rhodes
Cyril
Damien Tournoud
Dan Schmidt
Daniel Hollas
Daniel Knittl-Frank
Daniel Shahaf
Daniel Sturm
Daniele Tricoli
Daniil Larionov
Danny Lin
David Rogers
Davide Angelocola
Denis Savitskiy
Dexter
Dexter Morganov
DiamondeX
Dieter Ziller
Dino Karic
Dmitri Tikhonov
Dmitriy Smirnov
Doug Richardson
Duncan Dean
Dustin Frank
Ed Flanagan
Eden Hochbaum
Eduard Bardají Puig
Eric Henziger
Explorare
Ezra Buehler
Fabien-jrt
Fady Nagh
Felix Nehrke
Filip Kucharczyk
Fornost461
Frank
Frederico Mazzone
Frej Drejhammar
Guthrie McAfee Armstrong
HairyFotr
Hamid Nazari
Masood Fallahpoor
Mathieu Dubreuilh
Matt Cooper
Matt Trzcinski
Matthew Miner
Matthieu Moy
Mavaddat Javid
Max Coplan
Michael MacAskill
Michael Sheaver
Michael Welch
Michiel van der Wulp
Miguel Bernabeu
Mike Charles
Mike Pennisi
Mike Thibodeau
Mikhail Menshikov
Mitsuru Kariya
Máximo Cuadros
Niels Widger
Niko Stotz
Nils Reuße
Noelle Leigh
OliverSieweke
Olleg Samoylov
Osman Khwaja
Otto Kekäläinen
Owen
Pablo Schläpfer
Pascal Berger
Pascal Borreli
Patrice Krakow
Patrick Steinhardt
Pavel Janík
Paweł Krupiński
Pessimist
Peter Kokot
Petr Bodnar
Petr Janeček
Petr Kajzar
Phil Mitchell
Philippe Blain
Philippe Miossec
Pratik Nadagouda
Rafi
Raphael R
Ray Chen
Rex Kerr
Reza Ahmadi
Richard Hoyle
Ricky Senft
Volker-Weissmann
Wesley Gonçalves
William Gathoye
William Turrell
Wlodek Bzyl
Xavier Bonaventura
Y. E
Yann Soubeyrand
Your Name
Yue Lin Ho
Yuhang Guo
Yunhai Luo
Yusuke SATO
agkhall
ajax333221
alex-koziell
allen joslin
andreas
applecuckoo
atalakam
axmbo
bermudi
bripmccann
brotherben
delta4d
devwebcl
dualsky
evanderiel
eyherabh
flip111
flyingzumwalt
franjozen
goekboet
grgbnc
haripetrov
i-give-up
iprok
jingsam
jliljekrantz
johnhar
leerg
maks
mmikeww
mosdalsvsocld
nicktime
noureddin
patrick96
paveljanik
pedrorijo91
peterwwillis
petsuter

Hamidreza Mahdavipanah
Haruo Nakayama
Helmut K. C. Tessarek
Hemant Kumar Meena
Hidde de Vries
HonkingGoose
Howard
Ignacy
Igor
Ilker Cat
Jan Groenewald
Jannick Kremer
Jaswinder Singh
Jean-Noël Avila
Jeroen Oortwijn
Jim Hill
Jin Park
Joel Davies
Johannes Dewender
Rintze M. Zelle
Rob Blanco
Robert P. Goldman
Robert P. J. Day
Robert Theis
Rohan D'Souza
Roman Kosenko
Ronald Wampler
Rory
Ryan Cavicchioni
Rüdiger Herrmann
SATO Yusuke
Sam Ford
Sam Joseph
Sanders Kleinfeld
Sarah Schneider
Saurav Sachidanand
Scott Bronson
Scott Jones
rahrah
rmzelle
root
sanders@oreilly.com
sharpiro
slavos1
spacewander
td2014
twekberg
uerdogan
ugultopu
un1versal
xJom
xtreak
yakirwin
z-hed
zwPapEr
□□□□□□□
□□

Introduction

You're about to spend several hours of your life reading about Git. Let's take a minute to explain what we have in store for you. Here is a quick summary of the ten chapters and three appendices of this book.

In **Chapter 1**, we're going to cover Version Control Systems (VCSs) and Git basics — no technical stuff, just what Git is, why it came about in a land full of VCSs, what sets it apart, and why so many people are using it. Then, we'll explain how to download Git and set it up for the first time if you don't already have it on your system.

In **Chapter 2**, we will go over basic Git usage — how to use Git in the 80% of cases you'll encounter most often. After reading this chapter, you should be able to clone a repository, see what has happened in the history of the project, modify files, and contribute changes. If the book spontaneously combusts at this point, you should already be pretty useful wielding Git in the time it takes you to go pick up another copy.

Chapter 3 is about the branching model in Git, often described as Git's killer feature. Here you'll learn what truly sets Git apart from the pack. When you're done, you may feel the need to spend a quiet moment pondering how you lived before Git branching was part of your life.

Chapter 4 will cover Git on the server. This chapter is for those of you who want to set up Git inside your organization or on your own personal server for collaboration. We will also explore various hosted options if you prefer to let someone else handle that for you.

Chapter 5 will go over in full detail various distributed workflows and how to accomplish them with Git. When you are done with this chapter, you should be able to work expertly with multiple remote repositories, use Git over email and deftly juggle numerous remote branches and contributed patches.

Chapter 6 covers the GitHub hosting service and tooling in depth. We cover signing up for and managing an account, creating and using Git repositories, common workflows to contribute to projects and to accept contributions to yours, GitHub's programmatic interface and lots of little tips to make your life easier in general.

Chapter 7 is about advanced Git commands. Here you will learn about topics like mastering the scary 'reset' command, using binary search to identify bugs, editing history, revision selection in detail, and a lot more. This chapter will round out your knowledge of Git so that you are truly a master.

Chapter 8 is about configuring your custom Git environment. This includes setting up hook scripts to enforce or encourage customized policies and using environment configuration settings so you can work the way you want to. We will also cover building your own set of scripts to enforce a custom committing policy.

Chapter 9 deals with Git and other VCSs. This includes using Git in a Subversion (SVN) world and converting projects from other VCSs to Git. A lot of organizations still use SVN and are not about to change, but by this point you'll have learned the incredible power of Git — and this chapter shows you how to cope if you still have to use a SVN server. We also cover how to import projects from

several different systems in case you do convince everyone to make the plunge.

Chapter 10 delves into the murky yet beautiful depths of Git internals. Now that you know all about Git and can wield it with power and grace, you can move on to discuss how Git stores its objects, what the object model is, details of packfiles, server protocols, and more. Throughout the book, we will refer to sections of this chapter in case you feel like diving deep at that point; but if you are like us and want to dive into the technical details, you may want to read Chapter 10 first. We leave that up to you.

In **Appendix A**, we look at a number of examples of using Git in various specific environments. We cover a number of different GUIs and IDE programming environments that you may want to use Git in and what is available for you. If you're interested in an overview of using Git in your shell, your IDE, or your text editor, take a look here.

In **Appendix B**, we explore scripting and extending Git through tools like libgit2 and JGit. If you're interested in writing complex and fast custom tools and need low-level Git access, this is where you can see what that landscape looks like.

Finally, in **Appendix C**, we go through all the major Git commands one at a time and review where in the book we covered them and what we did with them. If you want to know where in the book we used any specific Git command you can look that up here.

Let's get started.

Getting Started

This chapter will be about getting started with Git. We will begin by explaining some background on version control tools, then move on to how to get Git running on your system and finally how to get it set up to start working with. At the end of this chapter you should understand why Git is around, why you should use it and you should be all set up to do so.

About Version Control

What is “version control”, and why should you care? Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later. For the examples in this book, you will use software source code as the files being version controlled, though in reality you can do this with nearly any type of file on a computer.

If you are a graphic or web designer and want to keep every version of an image or layout (which you would most certainly want to), a Version Control System (VCS) is a very wise thing to use. It allows you to revert selected files back to a previous state, revert the entire project back to a previous state, compare changes over time, see who last modified something that might be causing a problem, who introduced an issue and when, and more. Using a VCS also generally means that if you screw things up or lose files, you can easily recover. In addition, you get all this for very little overhead.

Local Version Control Systems

Many people’s version-control method of choice is to copy files into another directory (perhaps a time-stamped directory, if they’re clever). This approach is very common because it is so simple, but it is also incredibly error prone. It is easy to forget which directory you’re in and accidentally write to the wrong file or copy over files you don’t mean to.

To deal with this issue, programmers long ago developed local VCSs that had a simple database that kept all the changes to files under revision control.

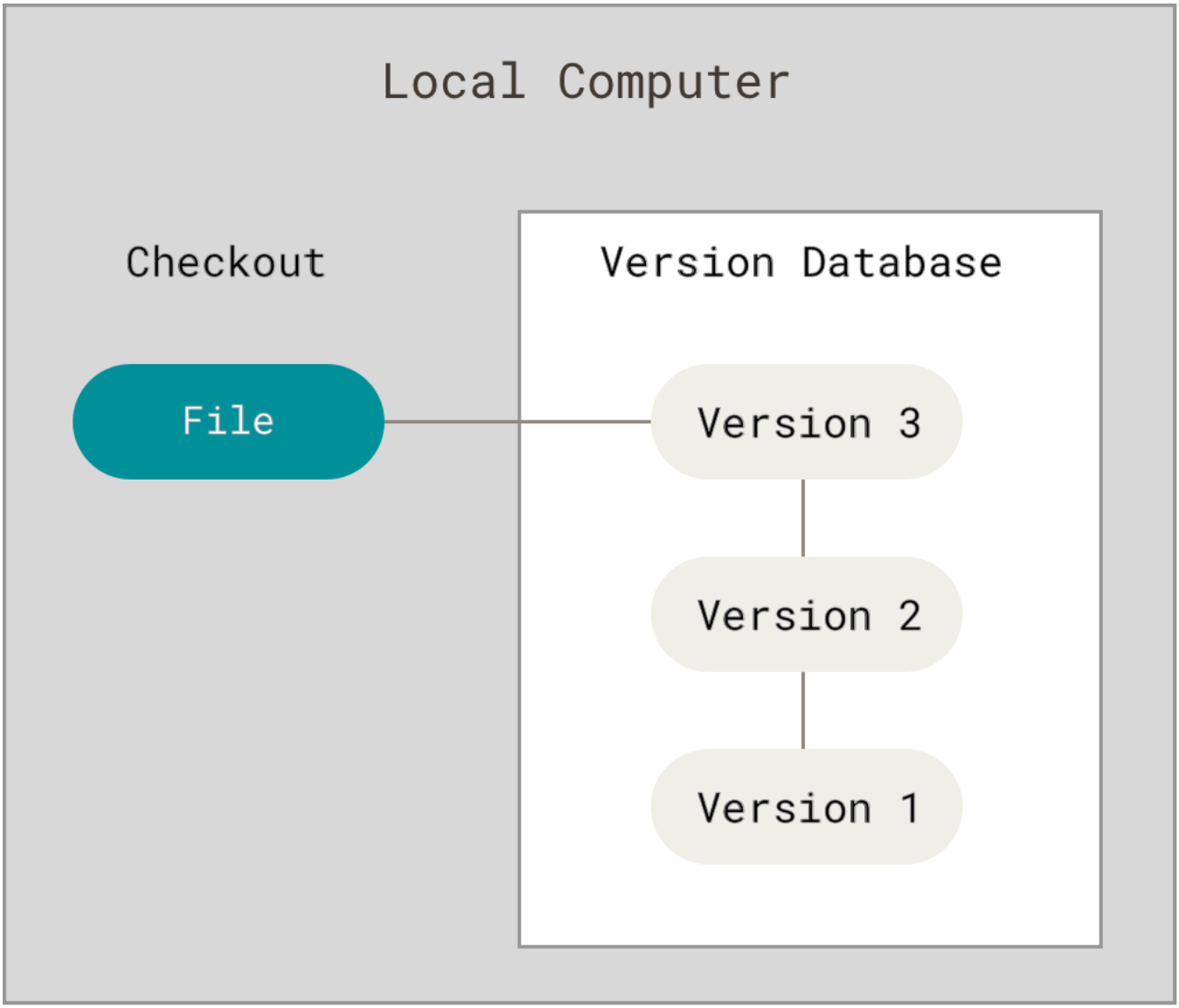

Figure 1. Local version control diagram

One of the most popular VCS tools was a system called RCS, which is still distributed with many computers today. RCS works by keeping patch sets (that is, the differences between files) in a special format on disk; it can then re-create what any file looked like at any point in time by adding up all the patches.

Centralized Version Control Systems

The next major issue that people encounter is that they need to collaborate with developers on other systems. To deal with this problem, Centralized Version Control Systems (CVCSs) were developed. These systems (such as CVS, Subversion, and Perforce) have a single server that contains all the versioned files, and a number of clients that check out files from that central place. For many years, this has been the standard for version control.

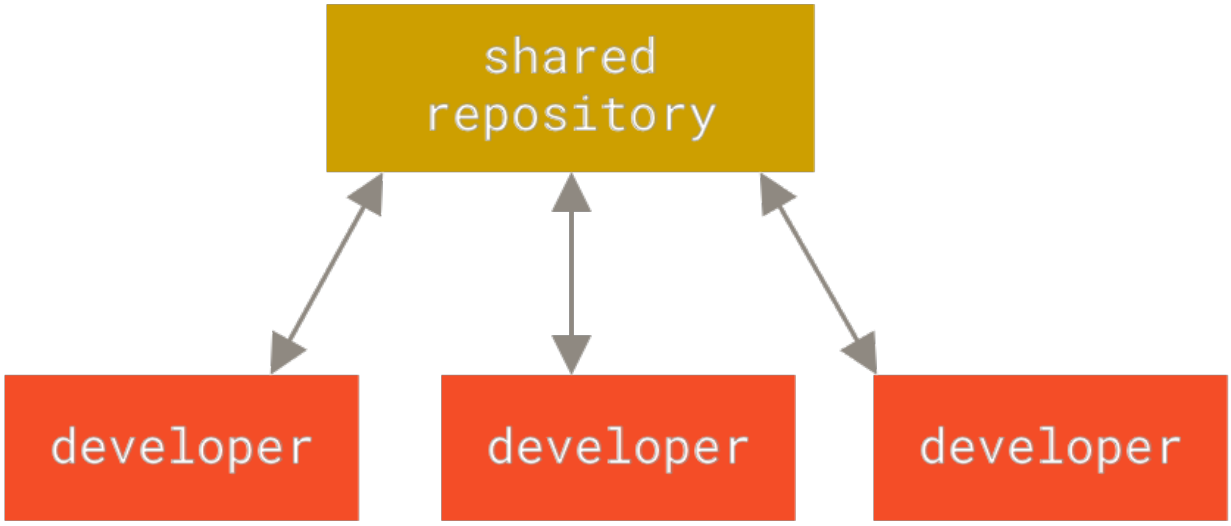

Figure 2. Centralized version control diagram

This setup offers many advantages, especially over local VCSs. For example, everyone knows to a certain degree what everyone else on the project is doing. Administrators have fine-grained control over who can do what, and it's far easier to administer a CVCS than it is to deal with local databases on every client.

However, this setup also has some serious downsides. The most obvious is the single point of failure that the centralized server represents. If that server goes down for an hour, then during that hour nobody can collaborate at all or save versioned changes to anything they're working on. If the hard disk the central database is on becomes corrupted, and proper backups haven't been kept, you lose absolutely everything — the entire history of the project except whatever single snapshots people happen to have on their local machines. Local VCSs suffer from this same problem — whenever you have the entire history of the project in a single place, you risk losing everything.

Distributed Version Control Systems

This is where Distributed Version Control Systems (DVCSs) step in. In a DVCS (such as Git, Mercurial or Darcs), clients don't just check out the latest snapshot of the files; rather, they fully mirror the repository, including its full history. Thus, if any server dies, and these systems were collaborating via that server, any of the client repositories can be copied back up to the server to restore it. Every clone is really a full backup of all the data.

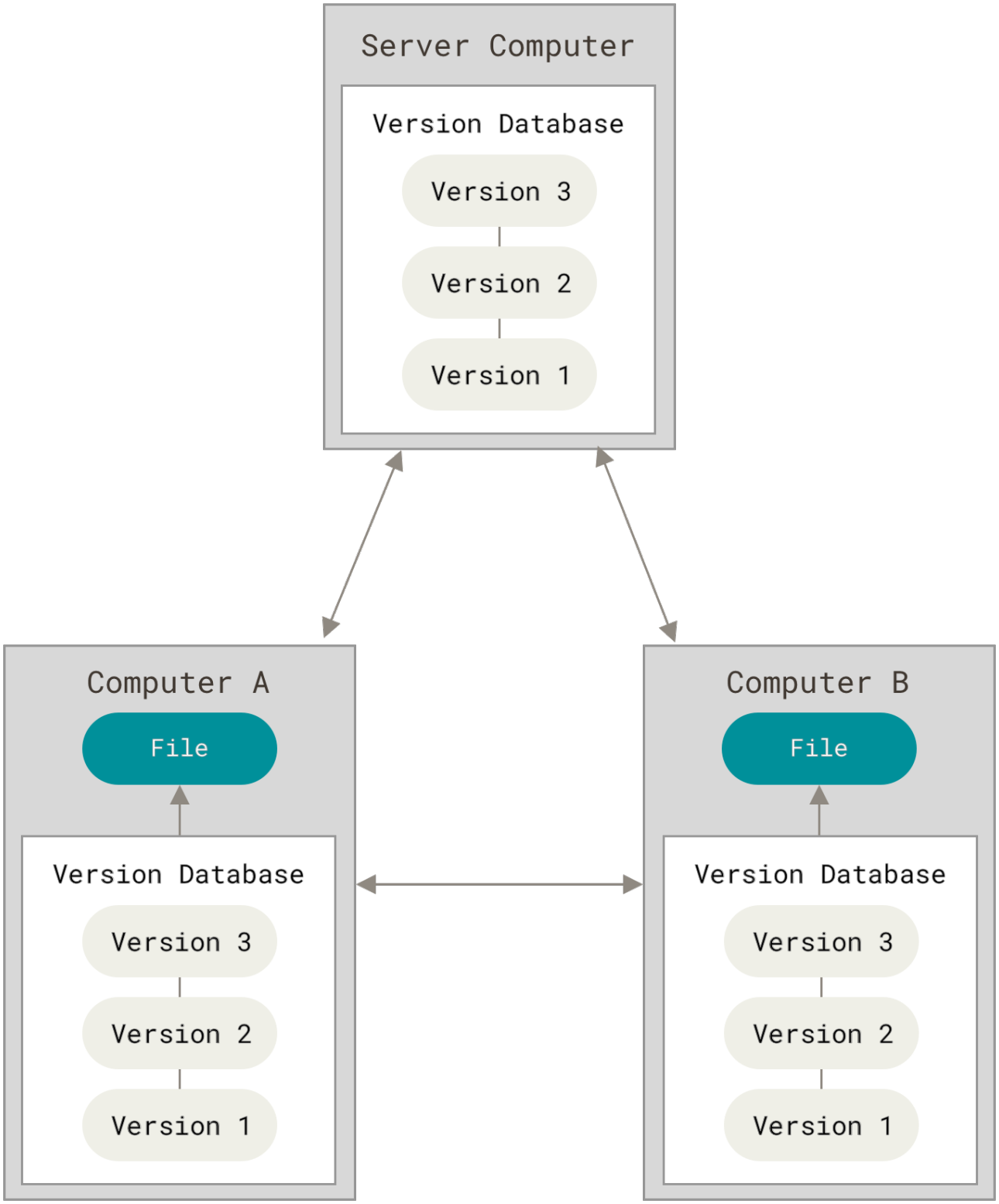

Figure 3. Distributed version control diagram

Furthermore, many of these systems deal pretty well with having several remote repositories they can work with, so you can collaborate with different groups of people in different ways simultaneously within the same project. This allows you to set up several types of workflows that aren't possible in centralized systems, such as hierarchical models.

A Short History of Git

As with many great things in life, Git began with a bit of creative destruction and fiery controversy.

The Linux kernel is an open source software project of fairly large scope. During the early years of the Linux kernel maintenance (1991–2002), changes to the software were passed around as patches and archived files. In 2002, the Linux kernel project began using a proprietary DVCS called BitKeeper.

In 2005, the relationship between the community that developed the Linux kernel and the commercial company that developed BitKeeper broke down, and the tool's free-of-charge status was revoked. This prompted the Linux development community (and in particular Linus Torvalds, the creator of Linux) to develop their own tool based on some of the lessons they learned while using BitKeeper. Some of the goals of the new system were as follows:

- Speed
- Simple design
- Strong support for non-linear development (thousands of parallel branches)
- Fully distributed
- Able to handle large projects like the Linux kernel efficiently (speed and data size)

Since its birth in 2005, Git has evolved and matured to be easy to use and yet retain these initial qualities. It's amazingly fast, it's very efficient with large projects, and it has an incredible branching system for non-linear development (see Git Branching).

What is Git?

So, what is Git in a nutshell? This is an important section to absorb, because if you understand what Git is and the fundamentals of how it works, then using Git effectively will probably be much easier for you. As you learn Git, try to clear your mind of the things you may know about other VCSs, such as CVS, Subversion or Perforce — doing so will help you avoid subtle confusion when using the tool. Even though Git's user interface is fairly similar to these other VCSs, Git stores and thinks about information in a very different way, and understanding these differences will help you avoid becoming confused while using it.

Snapshots, Not Differences

The major difference between Git and any other VCS (Subversion and friends included) is the way Git thinks about its data. Conceptually, most other systems store information as a list of file-based changes. These other systems (CVS, Subversion, Perforce, and so on) think of the information they store as a set of files and the changes made to each file over time (this is commonly described as *delta-based* version control).

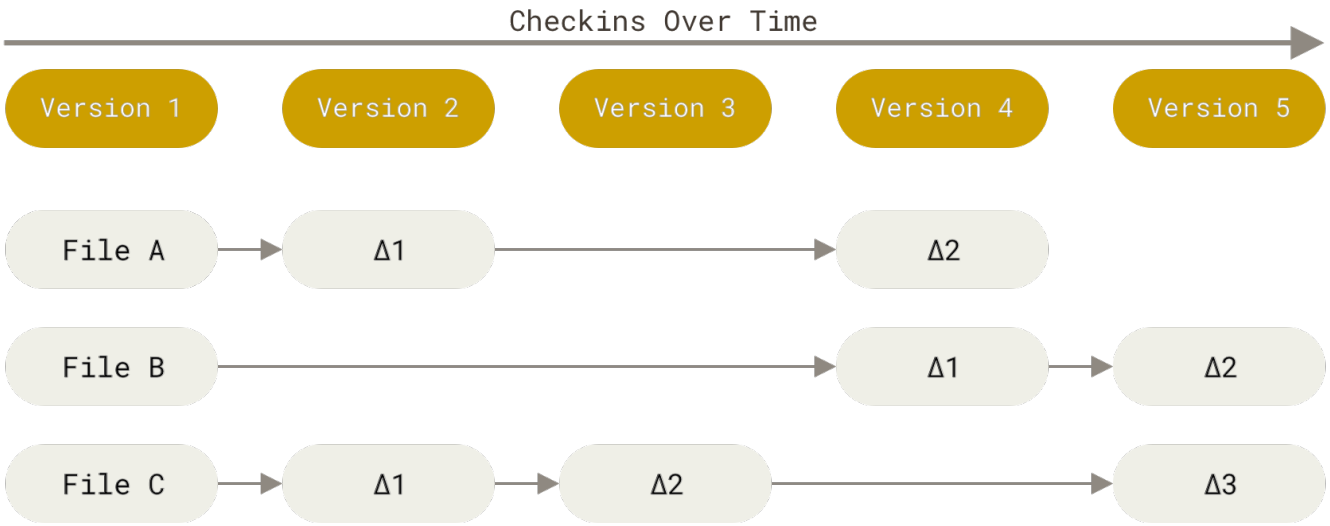

Figure 4. Storing data as changes to a base version of each file

Git doesn't think of or store its data this way. Instead, Git thinks of its data more like a series of snapshots of a miniature filesystem. With Git, every time you commit, or save the state of your project, Git basically takes a picture of what all your files look like at that moment and stores a reference to that snapshot. To be efficient, if files have not changed, Git doesn't store the file again, just a link to the previous identical file it has already stored. Git thinks about its data more like a **stream of snapshots**.

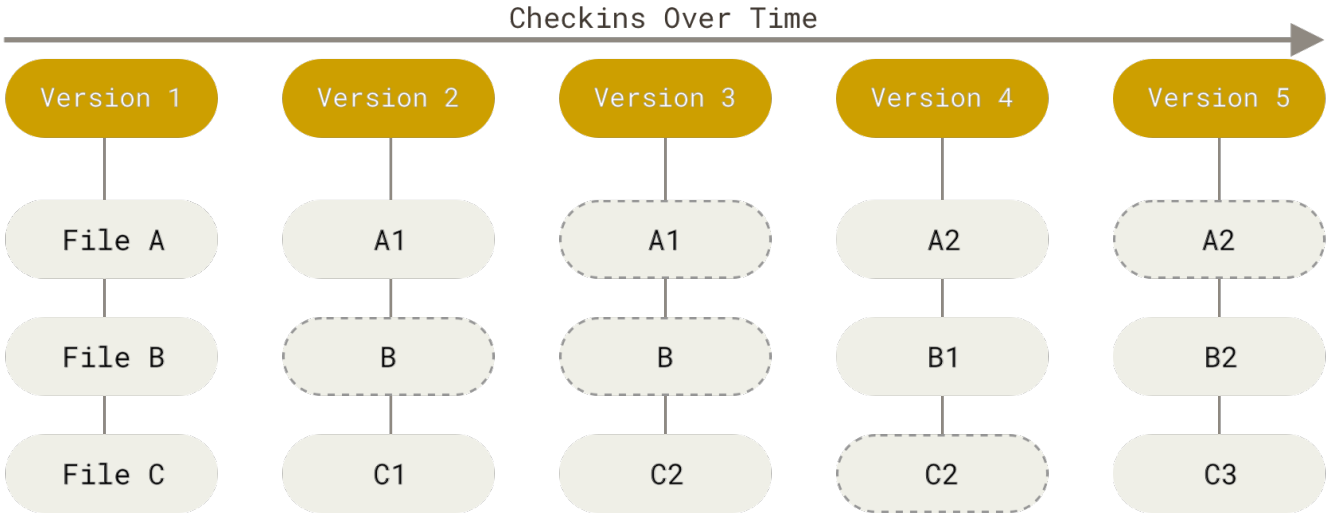

Figure 5. Storing data as snapshots of the project over time

This is an important distinction between Git and nearly all other VCSs. It makes Git reconsider almost every aspect of version control that most other systems copied from the previous generation. This makes Git more like a mini filesystem with some incredibly powerful tools built on top of it, rather than simply a VCS. We'll explore some of the benefits you gain by thinking of your data this way when we cover Git branching in Git Branching.

Nearly Every Operation Is Local

Most operations in Git need only local files and resources to operate — generally no information is needed from another computer on your network. If you're used to a CVCS where most operations have that network latency overhead, this aspect of Git will make you think that the gods of speed have blessed Git with unworldly powers. Because you have the entire history of the project right there on your local disk, most operations seem almost instantaneous.

For example, to browse the history of the project, Git doesn't need to go out to the server to get the history and display it for you — it simply reads it directly from your local database. This means you see the project history almost instantly. If you want to see the changes introduced between the current version of a file and the file a month ago, Git can look up the file a month ago and do a local difference calculation, instead of having to either ask a remote server to do it or pull an older version of the file from the remote server to do it locally.

This also means that there is very little you can't do if you're offline or off VPN. If you get on an airplane or a train and want to do a little work, you can commit happily (to your *local* copy, remember?) until you get to a network connection to upload. If you go home and can't get your VPN client working properly, you can still work. In many other systems, doing so is either impossible or painful. In Perforce, for example, you can't do much when you aren't connected to the server; in Subversion and CVS, you can edit files, but you can't commit changes to your database (because your database is offline). This may not seem like a huge deal, but you may be surprised what a big difference it can make.

Git Has Integrity

Everything in Git is checksummed before it is stored and is then referred to by that checksum. This means it's impossible to change the contents of any file or directory without Git knowing about it. This functionality is built into Git at the lowest levels and is integral to its philosophy. You can't lose information in transit or get file corruption without Git being able to detect it.

The mechanism that Git uses for this checksumming is called a SHA-1 hash. This is a 40-character string composed of hexadecimal characters (0–9 and a–f) and calculated based on the contents of a file or directory structure in Git. A SHA-1 hash looks something like this:

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

You will see these hash values all over the place in Git because it uses them so much. In fact, Git stores everything in its database not by file name but by the hash value of its contents.

Git Generally Only Adds Data

When you do actions in Git, nearly all of them only *add* data to the Git database. It is hard to get the system to do anything that is not undoable or to make it erase data in any way. As with any VCS, you can lose or mess up changes you haven't committed yet, but after you commit a snapshot into Git, it is very difficult to lose, especially if you regularly push your database to another repository.

This makes using Git a joy because we know we can experiment without the danger of severely screwing things up. For a more in-depth look at how Git stores its data and how you can recover data that seems lost, see Undoing Things.

The Three States

Pay attention now — here is the main thing to remember about Git if you want the rest of your learning process to go smoothly. Git has three main states that your files can reside in: *modified*, *staged*, and *committed*:

- Modified means that you have changed the file but have not committed it to your database yet.
- Staged means that you have marked a modified file in its current version to go into your next commit snapshot.
- Committed means that the data is safely stored in your local database.

This leads us to the three main sections of a Git project: the working tree, the staging area, and the Git directory.

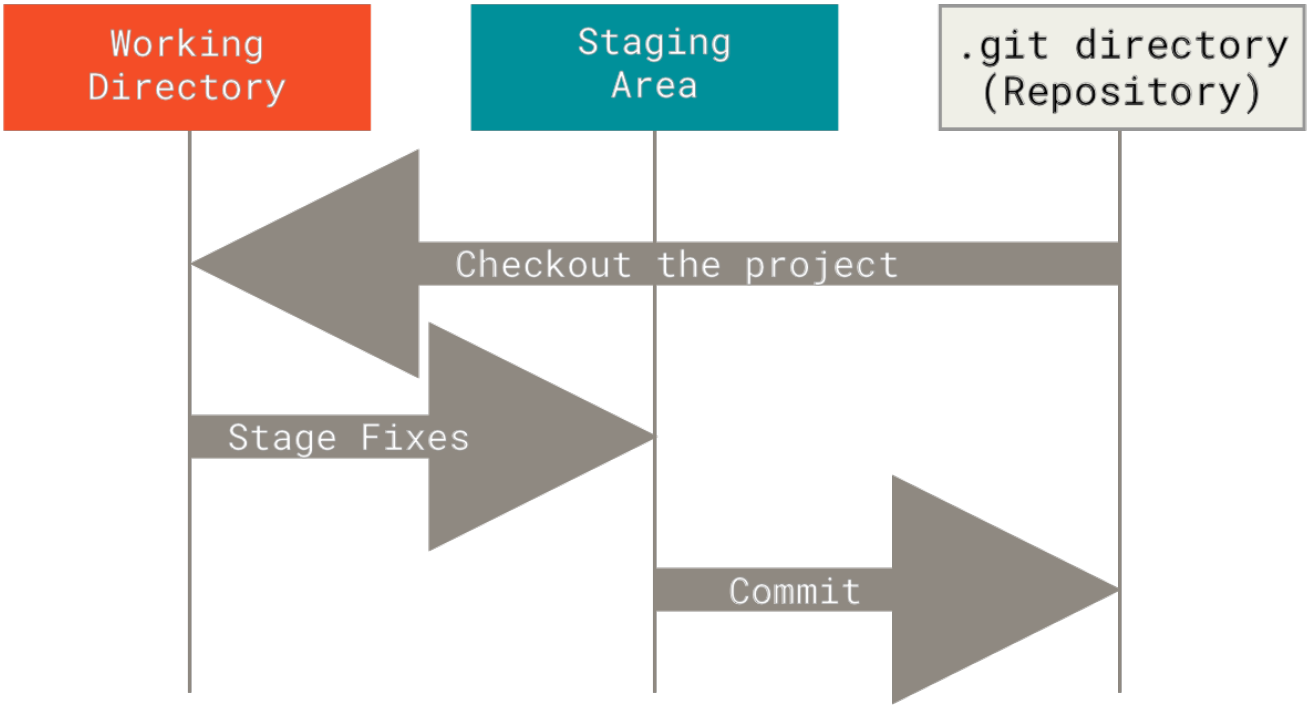

Figure 6. Working tree, staging area, and Git directory

The working tree is a single checkout of one version of the project. These files are pulled out of the compressed database in the Git directory and placed on disk for you to use or modify.

The staging area is a file, generally contained in your Git directory, that stores information about what will go into your next commit. Its technical name in Git parlance is the “index”, but the phrase “staging area” works just as well.

The Git directory is where Git stores the metadata and object database for your project. This is the most important part of Git, and it is what is copied when you *clone* a repository from another computer.

The basic Git workflow goes something like this:

1. You modify files in your working tree.
2. You selectively stage just those changes you want to be part of your next commit, which adds *only* those changes to the staging area.
3. You do a commit, which takes the files as they are in the staging area and stores that snapshot permanently to your Git directory.

If a particular version of a file is in the Git directory, it’s considered *committed*. If it has been

modified and was added to the staging area, it is *staged*. And if it was changed since it was checked out but has not been staged, it is *modified*. In Git Basics, you'll learn more about these states and how you can either take advantage of them or skip the staged part entirely.

The Command Line

There are a lot of different ways to use Git. There are the original command-line tools, and there are many graphical user interfaces of varying capabilities. For this book, we will be using Git on the command line. For one, the command line is the only place you can run *all* Git commands — most of the GUIs implement only a partial subset of Git functionality for simplicity. If you know how to run the command-line version, you can probably also figure out how to run the GUI version, while the opposite is not necessarily true. Also, while your choice of graphical client is a matter of personal taste, *all* users will have the command-line tools installed and available.

So we will expect you to know how to open Terminal in macOS or Command Prompt or PowerShell in Windows. If you don't know what we're talking about here, you may need to stop and research that quickly so that you can follow the rest of the examples and descriptions in this book.

Installing Git

Before you start using Git, you have to make it available on your computer. Even if it's already installed, it's probably a good idea to update to the latest version. You can either install it as a package or via another installer, or download the source code and compile it yourself.

This book was written using Git version 2. Since Git is quite excellent at preserving backwards compatibility, any recent version should work just fine. Though most of the commands we use should work even in ancient versions of Git, some of them might not or might act slightly differently.

Installing on Linux

If you want to install the basic Git tools on Linux via a binary installer, you can generally do so through the package management tool that comes with your distribution. If you're on Fedora (or any closely-related RPM-based distribution, such as RHEL or CentOS), you can use `dnf`:

```
$ sudo dnf install git-all
```

If you're on a Debian-based distribution, such as Ubuntu, try `apt`:

```
$ sudo apt install git-all
```

For more options, there are instructions for installing on several different Unix distributions on the Git website, at https://git-scm.com/download/linux.

Installing on macOS

There are several ways to install Git on macOS. The easiest is probably to install the Xcode Command Line Tools. On Mavericks (10.9) or above you can do this simply by trying to run `git` from the Terminal the very first time.

```
$ git --version
```

If you don't have it installed already, it will prompt you to install it.

If you want a more up to date version, you can also install it via a binary installer. A macOS Git installer is maintained and available for download at the Git website, at https://git-scm.com/download/mac.

Figure 7. Git macOS installer

Installing on Windows

There are also a few ways to install Git on Windows. The most official build is available for download on the Git website. Just go to https://git-scm.com/download/win and the download will start automatically. Note that this is a project called Git for Windows, which is separate from Git itself; for more information on it, go to https://gitforwindows.org.

To get an automated installation you can use the Git Chocolatey package. Note that the Chocolatey package is community maintained.

Installing from Source

Some people may instead find it useful to install Git from source, because you'll get the most recent version. The binary installers tend to be a bit behind, though as Git has matured in recent years, this has made less of a difference.

If you do want to install Git from source, you need to have the following libraries that Git depends on: autotools, curl, zlib, openssl, expat, and libiconv. For example, if you're on a system that has `dnf` (such as Fedora) or `apt-get` (such as a Debian-based system), you can use one of these commands to install the minimal dependencies for compiling and installing the Git binaries:

```
$ sudo dnf install dh-autoreconf curl-devel expat-devel gettext-devel \
  openssl-devel perl-devel zlib-devel
$ sudo apt-get install dh-autoreconf libcurl4-gnutls-dev libexpat1-dev \
  gettext libz-dev libssl-dev
```

In order to be able to add the documentation in various formats (doc, html, info), these additional dependencies are required:

```
$ sudo dnf install asciidoc xmlto docbook2X
$ sudo apt-get install asciidoc xmlto docbook2x
```

Users of RHEL and RHEL-derivatives like CentOS and Scientific Linux will have to enable the EPEL repository to download the `docbook2X` package.

If you're using a Debian-based distribution (Debian/Ubuntu/Ubuntu-derivatives), you also need the `install-info` package:

```
$ sudo apt-get install install-info
```

If you're using a RPM-based distribution (Fedora/RHEL/RHEL-derivatives), you also need the `getopt` package (which is already installed on a Debian-based distro):

```
$ sudo dnf install getopt
```

Additionally, if you're using Fedora/RHEL/RHEL-derivatives, you need to do this:

```
$ sudo ln -s /usr/bin/db2x_docbook2texi /usr/bin/docbook2x-texi
```

due to binary name differences.

When you have all the necessary dependencies, you can go ahead and grab the latest tagged release tarball from several places. You can get it via the kernel.org site, at https://www.kernel.org/pub/software/scm/git, or the mirror on the GitHub website, at https://github.com/git/git/tags. It's

generally a little clearer what the latest version is on the GitHub page, but the kernel.org page also has release signatures if you want to verify your download.

Then, compile and install:

```
$ tar -zxf git-2.8.0.tar.gz
$ cd git-2.8.0
$ make configure
$ ./configure --prefix=/usr
$ make all doc info
$ sudo make install install-doc install-html install-info
```

After this is done, you can also get Git via Git itself for updates:

```
$ git clone https://git.kernel.org/pub/scm/git/git.git
```

First-Time Git Setup

Now that you have Git on your system, you'll want to do a few things to customize your Git environment. You should have to do these things only once on any given computer; they'll stick around between upgrades. You can also change them at any time by running through the commands again.

Git comes with a tool called `git config` that lets you get and set configuration variables that control all aspects of how Git looks and operates. These variables can be stored in three different places:

1. `[path]/etc/gitconfig` file: Contains values applied to every user on the system and all their repositories. If you pass the option `--system` to `git config`, it reads and writes from this file specifically. Because this is a system configuration file, you would need administrative or superuser privilege to make changes to it.
2. `~/.gitconfig` or `~/.config/git/config` file: Values specific personally to you, the user. You can make Git read and write to this file specifically by passing the `--global` option, and this affects *all* of the repositories you work with on your system.
3. `config` file in the Git directory (that is, `.git/config`) of whatever repository you're currently using: Specific to that single repository. You can force Git to read from and write to this file with the `--local` option, but that is in fact the default. Unsurprisingly, you need to be located somewhere in a Git repository for this option to work properly.

Each level overrides values in the previous level, so values in `.git/config` trump those in `[path]/etc/gitconfig`.

On Windows systems, Git looks for the `.gitconfig` file in the `$HOME` directory (`C:\Users\$USER` for most people). It also still looks for `[path]/etc/gitconfig`, although it's relative to the MSys root, which is wherever you decide to install Git on your Windows system when you run the installer. If you are using version 2.x or later of Git for Windows, there is also a system-level config file at `C:\Documents and Settings\All Users\Application Data\Git\config` on Windows XP, and in

`C:\ProgramData\Git\config` on Windows Vista and newer. This config file can only be changed by `git config -f <file>` as an admin.

You can view all of your settings and where they are coming from using:

```
$ git config --list --show-origin
```

Your Identity

The first thing you should do when you install Git is to set your user name and email address. This is important because every Git commit uses this information, and it's immutably baked into the commits you start creating:

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

Again, you need to do this only once if you pass the `--global` option, because then Git will always use that information for your user on that system. If you want to override this with a different name or email address for specific projects, you can run the command without the `--global` option when you're in that project.

Many of the GUI tools will help you do this when you first run them.

Your Editor

Now that your identity is set up, you can configure the default text editor that will be used when Git needs you to type in a message. If not configured, Git uses your system's default editor.

If you want to use a different text editor, such as Emacs, you can do the following:

```
$ git config --global core.editor emacs
```

On a Windows system, if you want to use a different text editor, you must specify the full path to its executable file. This can be different depending on how your editor is packaged.

In the case of Notepad++, a popular programming editor, you are likely to want to use the 32-bit version, since at the time of writing the 64-bit version doesn't support all plug-ins. If you are on a 32-bit Windows system, or you have a 64-bit editor on a 64-bit system, you'll type something like this:

```
$ git config --global core.editor "'C:/Program Files/Notepad++/notepad++.exe'
-multiInst -notabbar -nosession -noPlugin"
```

Vim, Emacs and Notepad++ are popular text editors often used by developers on Unix-based systems like Linux and macOS or a Windows system. If you are using

another editor, or a 32-bit version, please find specific instructions for how to set up your favorite editor with Git in git config core.editor commands.

You may find, if you don't setup your editor like this, you get into a really confusing state when Git attempts to launch it. An example on a Windows system may include a prematurely terminated Git operation during a Git initiated edit.

Your default branch name

By default Git will create a branch called *master* when you create a new repository with `git init`. From Git version 2.28 onwards, you can set a different name for the initial branch.

To set *main* as the default branch name do:

```
$ git config --global init.defaultBranch main
```

Checking Your Settings

If you want to check your configuration settings, you can use the `git config --list` command to list all the settings Git can find at that point:

```
$ git config --list
user.name=John Doe
user.email=johndoe@example.com
color.status=auto
color.branch=auto
color.interactive=auto
color.diff=auto
...
```

You may see keys more than once, because Git reads the same key from different files (`[path]/etc/gitconfig` and `~/.gitconfig`, for example). In this case, Git uses the last value for each unique key it sees.

You can also check what Git thinks a specific key's value is by typing `git config <key>`:

```
$ git config user.name
John Doe
```

Since Git might read the same configuration variable value from more than one file, it's possible that you have an unexpected value for one of these values and you don't know why. In cases like that, you can query Git as to the *origin* for that value, and it will tell you which configuration file had the final say in setting that value:

```
$ git config --show-origin rerere.autoUpdate
file:/home/johndoe/.gitconfig   false
```

Getting Help

If you ever need help while using Git, there are three equivalent ways to get the comprehensive manual page (manpage) help for any of the Git commands:

```
$ git help <verb>
$ git <verb> --help
$ man git-<verb>
```

For example, you can get the manpage help for the `git config` command by running this:

```
$ git help config
```

These commands are nice because you can access them anywhere, even offline. If the manpages and this book aren't enough and you need in-person help, you can try the `#git`, `#github`, or `#gitlab` channels on the Libera Chat IRC server, which can be found at https://libera.chat/. These channels are regularly filled with hundreds of people who are all very knowledgeable about Git and are often willing to help.

In addition, if you don't need the full-blown manpage help, but just need a quick refresher on the available options for a Git command, you can ask for the more concise "help" output with the `-h` option, as in:

```
$ git add -h
usage: git add [<options>] [--] <pathspec>...

    -n, --dry-run               dry run
    -v, --verbose               be verbose

    -i, --interactive           interactive picking
    -p, --patch                 select hunks interactively
    -e, --edit                  edit current diff and apply
    -f, --force                 allow adding otherwise ignored files
    -u, --update                update tracked files
    --renormalize               renormalize EOL of tracked files (implies -u)
    -N, --intent-to-add         record only the fact that the path will be added later
    -A, --all                   add changes from all tracked and untracked files
    --ignore-removal            ignore paths removed in the working tree (same as --no
-all)
    --refresh                   don't add, only refresh the index
    --ignore-errors             just skip files which cannot be added because of
errors
```

```
    --ignore-missing           check if - even missing - files are ignored in dry run
    --sparse                   allow updating entries outside of the sparse-checkout
cone
    --chmod (+|-)x             override the executable bit of the listed files
    --pathspec-from-file <file> read pathspec from file
    --pathspec-file-nul        with --pathspec-from-file, pathspec elements are
separated with NUL character
```

Summary

You should have a basic understanding of what Git is and how it's different from any centralized version control systems you may have been using previously. You should also now have a working version of Git on your system that's set up with your personal identity. It's now time to learn some Git basics.

Git Basics

If you can read only one chapter to get going with Git, this is it. This chapter covers every basic command you need to do the vast majority of the things you'll eventually spend your time doing with Git. By the end of the chapter, you should be able to configure and initialize a repository, begin and stop tracking files, and stage and commit changes. We'll also show you how to set up Git to ignore certain files and file patterns, how to undo mistakes quickly and easily, how to browse the history of your project and view changes between commits, and how to push and pull from remote repositories.

Getting a Git Repository

You typically obtain a Git repository in one of two ways:

1. You can take a local directory that is currently not under version control, and turn it into a Git repository, or
2. You can *clone* an existing Git repository from elsewhere.

In either case, you end up with a Git repository on your local machine, ready for work.

Initializing a Repository in an Existing Directory

If you have a project directory that is currently not under version control and you want to start controlling it with Git, you first need to go to that project's directory. If you've never done this, it looks a little different depending on which system you're running:

for Linux:

```
$ cd /home/user/my_project
```

for macOS:

```
$ cd /Users/user/my_project
```

for Windows:

```
$ cd C:/Users/user/my_project
```

and type:

```
$ git init
```

This creates a new subdirectory named `.git` that contains all of your necessary repository files — a Git repository skeleton. At this point, nothing in your project is tracked yet. See Git Internals for

more information about exactly what files are contained in the `.git` directory you just created.

If you want to start version-controlling existing files (as opposed to an empty directory), you should probably begin tracking those files and do an initial commit. You can accomplish that with a few `git add` commands that specify the files you want to track, followed by a `git commit`:

```
$ git add *.c
$ git add LICENSE
$ git commit -m 'Initial project version'
```

We'll go over what these commands do in just a minute. At this point, you have a Git repository with tracked files and an initial commit.

Cloning an Existing Repository

If you want to get a copy of an existing Git repository — for example, a project you'd like to contribute to — the command you need is `git clone`. If you're familiar with other VCSs such as Subversion, you'll notice that the command is "clone" and not "checkout". This is an important distinction — instead of getting just a working copy, Git receives a full copy of nearly all data that the server has. Every version of every file for the history of the project is pulled down by default when you run `git clone`. In fact, if your server disk gets corrupted, you can often use nearly any of the clones on any client to set the server back to the state it was in when it was cloned (you may lose some server-side hooks and such, but all the versioned data would be there — see Getting Git on a Server for more details).

You clone a repository with `git clone <url>`. For example, if you want to clone the Git linkable library called `libgit2`, you can do so like this:

```
$ git clone https://github.com/libgit2/libgit2
```

That creates a directory named `libgit2`, initializes a `.git` directory inside it, pulls down all the data for that repository, and checks out a working copy of the latest version. If you go into the new `libgit2` directory that was just created, you'll see the project files in there, ready to be worked on or used.

If you want to clone the repository into a directory named something other than `libgit2`, you can specify the new directory name as an additional argument:

```
$ git clone https://github.com/libgit2/libgit2 mylibgit
```

That command does the same thing as the previous one, but the target directory is called `mylibgit`.

Git has a number of different transfer protocols you can use. The previous example uses the `https://` protocol, but you may also see `git://` or `user@server:path/to/repo.git`, which uses the SSH transfer protocol. Getting Git on a Server will introduce all of the available options the server can set up to access your Git repository and the pros and cons of each.

Recording Changes to the Repository

At this point, you should have a *bona fide* Git repository on your local machine, and a checkout or *working copy* of all of its files in front of you. Typically, you'll want to start making changes and committing snapshots of those changes into your repository each time the project reaches a state you want to record.

Remember that each file in your working directory can be in one of two states: *tracked* or *untracked*. Tracked files are files that were in the last snapshot, as well as any newly staged files; they can be unmodified, modified, or staged. In short, tracked files are files that Git knows about.

Untracked files are everything else — any files in your working directory that were not in your last snapshot and are not in your staging area. When you first clone a repository, all of your files will be tracked and unmodified because Git just checked them out and you haven't edited anything.

As you edit files, Git sees them as modified, because you've changed them since your last commit. As you work, you selectively stage these modified files and then commit all those staged changes, and the cycle repeats.

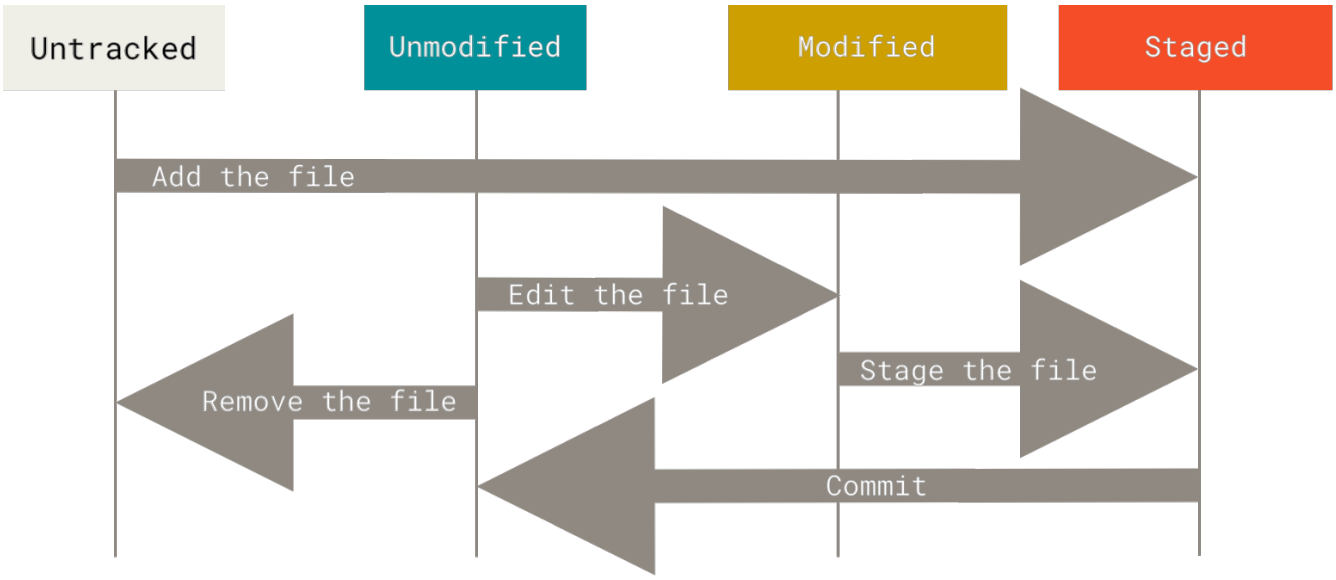

Figure 8. The lifecycle of the status of your files

Checking the Status of Your Files

The main tool you use to determine which files are in which state is the `git status` command. If you run this command directly after a clone, you should see something like this:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working tree clean
```

This means you have a clean working directory; in other words, none of your tracked files are modified. Git also doesn't see any untracked files, or they would be listed here. Finally, the command tells you which branch you're on and informs you that it has not diverged from the same

branch on the server. For now, that branch is always `master`, which is the default; you won't worry about it here. Git Branching will go over branches and references in detail.

GitHub changed the default branch name from `master` to `main` in mid-2020, and other Git hosts followed suit. So you may find that the default branch name in some newly created repositories is `main` and not `master`. In addition, the default branch name can be changed (as you have seen in Your default branch name), so you may see a different name for the default branch.

However, Git itself still uses `master` as the default, so we will use it throughout the book.

Let's say you add a new file to your project, a simple `README` file. If the file didn't exist before, and you run `git status`, you see your untracked file like so:

```
$ echo 'My Project' > README
$ 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)

    README

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

You can see that your new `README` file is untracked, because it's under the "Untracked files" heading in your status output. Untracked basically means that Git sees a file you didn't have in the previous snapshot (commit), and which hasn't yet been staged; Git won't start including it in your commit snapshots until you explicitly tell it to do so. It does this so you don't accidentally begin including generated binary files or other files that you did not mean to include. You do want to start including `README`, so let's start tracking the file.

Tracking New Files

In order to begin tracking a new file, you use the command `git add`. To begin tracking the `README` file, you can run this:

```
$ git add README
```

If you run your status command again, you can see that your `README` file is now tracked and staged to be committed:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
```

```
  (use "git restore --staged <file>..." to unstage)

    new file:   README
```

You can tell that it's staged because it's under the "Changes to be committed" heading. If you commit at this point, the version of the file at the time you ran `git add` is what will be in the subsequent historical snapshot. You may recall that when you ran `git init` earlier, you then ran `git add <files>` — that was to begin tracking files in your directory. The `git add` command takes a path name for either a file or a directory; if it's a directory, the command adds all the files in that directory recursively.

Staging Modified Files

Let's change a file that was already tracked. If you change a previously tracked file called `CONTRIBUTING.md` and then run your `git status` command again, you get something that looks like this:

```
$ 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:   README

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:   CONTRIBUTING.md
```

The `CONTRIBUTING.md` file appears under a section named "Changes not staged for commit" — which means that a file that is tracked has been modified in the working directory but not yet staged. To stage it, you run the `git add` command. `git add` is a multipurpose command — you use it to begin tracking new files, to stage files, and to do other things like marking merge-conflicted files as resolved. It may be helpful to think of it more as "add precisely this content to the next commit" rather than "add this file to the project". Let's run `git add` now to stage the `CONTRIBUTING.md` file, and then run `git status` again:

```
$ git add CONTRIBUTING.md
$ 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:   README
```

```
    modified:   CONTRIBUTING.md
```

Both files are staged and will go into your next commit. At this point, suppose you remember one little change that you want to make in `CONTRIBUTING.md` before you commit it. You open it again and make that change, and you're ready to commit. However, let's run `git status` one more time:

```
$ vim CONTRIBUTING.md
$ 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:   README
    modified:   CONTRIBUTING.md

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:   CONTRIBUTING.md
```

What the heck? Now `CONTRIBUTING.md` is listed as both staged *and* unstaged. How is that possible? It turns out that Git stages a file exactly as it is when you run the `git add` command. If you commit now, the version of `CONTRIBUTING.md` as it was when you last ran the `git add` command is how it will go into the commit, not the version of the file as it looks in your working directory when you run `git commit`. If you modify a file after you run `git add`, you have to run `git add` again to stage the latest version of the file:

```
$ git add CONTRIBUTING.md
$ 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:   README
    modified:   CONTRIBUTING.md
```

Short Status

While the `git status` output is pretty comprehensive, it's also quite wordy. Git also has a short status flag so you can see your changes in a more compact way. If you run `git status -s` or `git status --short` you get a far more simplified output from the command:

```
$ git status -s
 M README
```

```
MM Rakefile
A  lib/git.rb
M  lib/simplegit.rb
?? LICENSE.txt
```

New files that aren't tracked have a `??` next to them, new files that have been added to the staging area have an `A`, modified files have an `M` and so on. There are two columns to the output — the left-hand column indicates the status of the staging area and the right-hand column indicates the status of the working tree. So for example in that output, the `README` file is modified in the working directory but not yet staged, while the `lib/simplegit.rb` file is modified and staged. The `Rakefile` was modified, staged and then modified again, so there are changes to it that are both staged and unstaged.

Ignoring Files

Often, you'll have a class of files that you don't want Git to automatically add or even show you as being untracked. These are generally automatically generated files such as log files or files produced by your build system. In such cases, you can create a file listing patterns to match them named `.gitignore`. Here is an example `.gitignore` file:

```
$ cat .gitignore
*.[oa]
*~
```

The first line tells Git to ignore any files ending in ".o" or ".a" — object and archive files that may be the product of building your code. The second line tells Git to ignore all files whose names end with a tilde (`~`), which is used by many text editors such as Emacs to mark temporary files. You may also include a log, tmp, or pid directory; automatically generated documentation; and so on. Setting up a `.gitignore` file for your new repository before you get going is generally a good idea so you don't accidentally commit files that you really don't want in your Git repository.

The rules for the patterns you can put in the `.gitignore` file are as follows:

- Blank lines or lines starting with `#` are ignored.
- Standard glob patterns work, and will be applied recursively throughout the entire working tree.
- You can start patterns with a forward slash (`/`) to avoid recursivity.
- You can end patterns with a forward slash (`/`) to specify a directory.
- You can negate a pattern by starting it with an exclamation point (`!`).

Glob patterns are like simplified regular expressions that shells use. An asterisk (`*`) matches zero or more characters; `[abc]` matches any character inside the brackets (in this case a, b, or c); a question mark (`?`) matches a single character; and brackets enclosing characters separated by a hyphen (`[0-9]`) matches any character between them (in this case 0 through 9). You can also use two asterisks to match nested directories; `a/**/z` would match `a/z`, `a/b/z`, `a/b/c/z`, and so on.

Here is another example `.gitignore` file:

```
# ignore all .a files
*.a

# but do track lib.a, even though you're ignoring .a files above
!lib.a

# only ignore the TODO file in the current directory, not subdir/TODO
/TODO

# ignore all files in any directory named build
build/

# ignore doc/notes.txt, but not doc/server/arch.txt
doc/*.txt

# ignore all .pdf files in the doc/ directory and any of its subdirectories
doc/**/*.pdf
```

GitHub maintains a fairly comprehensive list of good `.gitignore` file examples for dozens of projects and languages at https://github.com/github/gitignore if you want a starting point for your project.

In the simple case, a repository might have a single `.gitignore` file in its root directory, which applies recursively to the entire repository. However, it is also possible to have additional `.gitignore` files in subdirectories. The rules in these nested `.gitignore` files apply only to the files under the directory where they are located. The Linux kernel source repository has 206 `.gitignore` files.

It is beyond the scope of this book to get into the details of multiple `.gitignore` files; see `man gitignore` for the details.

Viewing Your Staged and Unstaged Changes

If the `git status` command is too vague for you — you want to know exactly what you changed, not just which files were changed — you can use the `git diff` command. We'll cover `git diff` in more detail later, but you'll probably use it most often to answer these two questions: What have you changed but not yet staged? And what have you staged that you are about to commit? Although `git status` answers those questions very generally by listing the file names, `git diff` shows you the exact lines added and removed — the patch, as it were.

Let's say you edit and stage the `README` file again and then edit the `CONTRIBUTING.md` file without staging it. If you run your `git status` command, you once again see something like this:

```
$ 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:   README

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:   CONTRIBUTING.md
```

To see what you've changed but not yet staged, type `git diff` with no other arguments:

```
$ git diff
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 8ebb991..643e24f 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -65,7 +65,8 @@ branch directly, things can get messy.
 Please include a nice description of your changes when you submit your PR;
 if we have to read the whole diff to figure out why you're contributing
 in the first place, you're less likely to get feedback and have your change
-merged in.
+merged in. Also, split your changes into comprehensive chunks if your patch is
+longer than a dozen lines.

 If you are starting to work on a particular area, feel free to submit a PR
 that highlights your work in progress (and note in the PR title that it's
```

That command compares what is in your working directory with what is in your staging area. The result tells you the changes you've made that you haven't yet staged.

If you want to see what you've staged that will go into your next commit, you can use `git diff --staged`. This command compares your staged changes to your last commit:

```
$ git diff --staged
diff --git a/README b/README
new file mode 100644
index 0000000..03902a1
--- /dev/null
+++ b/README
@@ -0,0 +1 @@
+My Project
```

It's important to note that `git diff` by itself doesn't show all changes made since your last commit — only changes that are still unstaged. If you've staged all of your changes, `git diff` will give you no output.

For another example, if you stage the `CONTRIBUTING.md` file and then edit it, you can use `git diff` to see the changes in the file that are staged and the changes that are unstaged. If our environment looks like this:

```
$ git add CONTRIBUTING.md
$ echo '# test line' >> CONTRIBUTING.md
$ 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:   CONTRIBUTING.md

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:   CONTRIBUTING.md
```

Now you can use `git diff` to see what is still unstaged:

```
$ git diff
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 643e24f..87f08c8 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -119,3 +119,4 @@ at the
 ## Starter Projects

 See our [projects
list](https://github.com/libgit2/libgit2/blob/development/PROJECTS.md).
+# test line
```

and `git diff --cached` to see what you've staged so far (`--staged` and `--cached` are synonyms):

```
$ git diff --cached
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 8ebb991..643e24f 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -65,7 +65,8 @@ branch directly, things can get messy.
 Please include a nice description of your changes when you submit your PR;
 if we have to read the whole diff to figure out why you're contributing
 in the first place, you're less likely to get feedback and have your change
-merged in.
+merged in. Also, split your changes into comprehensive chunks if your patch is
+longer than a dozen lines.
```

```
If you are starting to work on a particular area, feel free to submit a PR
that highlights your work in progress (and note in the PR title that it's
```

Git Diff in an External Tool

We will continue to use the `git diff` command in various ways throughout the rest of the book. There is another way to look at these diffs if you prefer a graphical or external diff viewing program instead. If you run `git difftool` instead of `git diff`, you can view any of these diffs in software like emerge, vimdiff and many more (including commercial products). Run `git difftool --tool-help` to see what is available on your system.

Committing Your Changes

Now that your staging area is set up the way you want it, you can commit your changes. Remember that anything that is still unstaged — any files you have created or modified that you haven't run `git add` on since you edited them — won't go into this commit. They will stay as modified files on your disk. In this case, let's say that the last time you ran `git status`, you saw that everything was staged, so you're ready to commit your changes. The simplest way to commit is to type `git commit`:

```
$ git commit
```

Doing so launches your editor of choice.

This is set by your shell's `EDITOR` environment variable — usually vim or emacs, although you can configure it with whatever you want using the `git config --global core.editor` command as you saw in Getting Started.

The editor displays the following text (this example is a Vim screen):

```
# 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:
#   new file:   README
#   modified:   CONTRIBUTING.md
#
~
~
~
".git/COMMIT_EDITMSG" 9L, 283C
```

You can see that the default commit message contains the latest output of the `git status` command commented out and one empty line on top. You can remove these comments and type your commit

message, or you can leave them there to help you remember what you're committing.

For an even more explicit reminder of what you've modified, you can pass the `-v` option to `git commit`. Doing so also puts the diff of your change in the editor so you can see exactly what changes you're committing.

When you exit the editor, Git creates your commit with that commit message (with the comments and diff stripped out).

Alternatively, you can type your commit message inline with the `commit` command by specifying it after a `-m` flag, like this:

```
$ git commit -m "Story 182: fix benchmarks for speed"
[master 463dc4f] Story 182: fix benchmarks for speed
 2 files changed, 2 insertions(+)
 create mode 100644 README
```

Now you've created your first commit! You can see that the commit has given you some output about itself: which branch you committed to (`master`), what SHA-1 checksum the commit has (`463dc4f`), how many files were changed, and statistics about lines added and removed in the commit.

Remember that the commit records the snapshot you set up in your staging area. Anything you didn't stage is still sitting there modified; you can do another commit to add it to your history. Every time you perform a commit, you're recording a snapshot of your project that you can revert to or compare to later.

Skipping the Staging Area

Although it can be amazingly useful for crafting commits exactly how you want them, the staging area is sometimes a bit more complex than you need in your workflow. If you want to skip the staging area, Git provides a simple shortcut. Adding the `-a` option to the `git commit` command makes Git automatically stage every file that is already tracked before doing the commit, letting you skip the `git add` part:

```
$ 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:   CONTRIBUTING.md

no changes added to commit (use "git add" and/or "git commit -a")
$ git commit -a -m 'Add new benchmarks'
[master 83e38c7] Add new benchmarks
```

```
 1 file changed, 5 insertions(+), 0 deletions(-)
```

Notice how you don't have to run `git add` on the `CONTRIBUTING.md` file in this case before you commit. That's because the `-a` flag includes all changed files. This is convenient, but be careful; sometimes this flag will cause you to include unwanted changes.

Removing Files

To remove a file from Git, you have to remove it from your tracked files (more accurately, remove it from your staging area) and then commit. The `git rm` command does that, and also removes the file from your working directory so you don't see it as an untracked file the next time around.

If you simply remove the file from your working directory, it shows up under the "Changes not staged for commit" (that is, *unstaged*) area of your `git status` output:

```
$ rm PROJECTS.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        deleted:    PROJECTS.md

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

Then, if you run `git rm`, it stages the file's removal:

```
$ git rm PROJECTS.md
rm 'PROJECTS.md'
$ 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)

    deleted:    PROJECTS.md
```

The next time you commit, the file will be gone and no longer tracked. If you modified the file or had already added it to the staging area, you must force the removal with the `-f` option. This is a safety feature to prevent accidental removal of data that hasn't yet been recorded in a snapshot and that can't be recovered from Git.

Another useful thing you may want to do is to keep the file in your working tree but remove it from your staging area. In other words, you may want to keep the file on your hard drive but not have Git track it anymore. This is particularly useful if you forgot to add something to your `.gitignore`

file and accidentally staged it, like a large log file or a bunch of `.a` compiled files. To do this, use the `--cached` option:

```
$ git rm --cached README
```

You can pass files, directories, and file-glob patterns to the `git rm` command. That means you can do things such as:

```
$ git rm log/\*.log
```

Note the backslash (`\`) in front of the `*`. This is necessary because Git does its own filename expansion in addition to your shell's filename expansion. This command removes all files that have the `.log` extension in the `log/` directory. Or, you can do something like this:

```
$ git rm \*~
```

This command removes all files whose names end with a `~`.

Moving Files

Unlike many other VCSs, Git doesn't explicitly track file movement. If you rename a file in Git, no metadata is stored in Git that tells it you renamed the file. However, Git is pretty smart about figuring that out after the fact — we'll deal with detecting file movement a bit later.

Thus it's a bit confusing that Git has a `mv` command. If you want to rename a file in Git, you can run something like:

```
$ git mv file_from file_to
```

and it works fine. In fact, if you run something like this and look at the status, you'll see that Git considers it a renamed file:

```
$ git mv README.md README
$ 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)

    renamed:    README.md -> README
```

However, this is equivalent to running something like this:

```
$ mv README.md README
```

```
$ git rm README.md
$ git add README
```

Git figures out that it's a rename implicitly, so it doesn't matter if you rename a file that way or with the `mv` command. The only real difference is that `git mv` is one command instead of three — it's a convenience function. More importantly, you can use any tool you like to rename a file, and address the `add`/`rm` later, before you commit.

Viewing the Commit History

After you have created several commits, or if you have cloned a repository with an existing commit history, you'll probably want to look back to see what has happened. The most basic and powerful tool to do this is the `git log` command.

These examples use a very simple project called "simplegit". To get the project, run:

```
$ git clone https://github.com/schacon/simplegit-progit
```

When you run `git log` in this project, you should get output that looks something like this:

```
$ git log
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    Change version number

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    Remove unnecessary test

commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700

    Initial commit
```

By default, with no arguments, `git log` lists the commits made in that repository in reverse chronological order; that is, the most recent commits show up first. As you can see, this command lists each commit with its SHA-1 checksum, the author's name and email, the date written, and the commit message.

A huge number and variety of options to the `git log` command are available to show you exactly what you're looking for. Here, we'll show you some of the most popular.

One of the more helpful options is `-p` or `--patch`, which shows the difference (the *patch* output) introduced in each commit. You can also limit the number of log entries displayed, such as using `-2` to show only the last two entries.

```
$ git log -p -2
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    Change version number

diff --git a/Rakefile b/Rakefile
index a874b73..8f94139 100644
--- a/Rakefile
+++ b/Rakefile
@@ -5,7 +5,7 @@ require 'rake/gempackagetask'
 spec = Gem::Specification.new do |s|
     s.platform  =   Gem::Platform::RUBY
     s.name      =   "simplegit"
-    s.version   =   "0.1.0"
+    s.version   =   "0.1.1"
     s.author    =   "Scott Chacon"
     s.email     =   "schacon@gee-mail.com"
     s.summary   =   "A simple gem for using Git in Ruby code."

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    Remove unnecessary test

diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index a0a60ae..47c6340 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -18,8 +18,3 @@ class SimpleGit
     end

 end
-
-if $0 == __FILE__
-  git = SimpleGit.new
-  puts git.show
-end
```

This option displays the same information but with a diff directly following each entry. This is very helpful for code review or to quickly browse what happened during a series of commits that a collaborator has added. You can also use a series of summarizing options with `git log`. For example, if you want to see some abbreviated stats for each commit, you can use the `--stat` option:

```
$ git log --stat
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    Change version number

 Rakefile | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    Remove unnecessary test

 lib/simplegit.rb | 5 -----
 1 file changed, 5 deletions(-)

commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700

    Initial commit

 README           |  6 ++++++
 Rakefile         | 23 +++++++++++++++++++++++
 lib/simplegit.rb | 25 +++++++++++++++++++++++++
 3 files changed, 54 insertions(+)
```

As you can see, the `--stat` option prints below each commit entry a list of modified files, how many files were changed, and how many lines in those files were added and removed. It also puts a summary of the information at the end.

Another really useful option is `--pretty`. This option changes the log output to formats other than the default. A few prebuilt option values are available for you to use. The `oneline` value for this option prints each commit on a single line, which is useful if you're looking at a lot of commits. In addition, the `short`, `full`, and `fuller` values show the output in roughly the same format but with less or more information, respectively:

```
$ git log --pretty=oneline
ca82a6dff817ec66f44342007202690a93763949 Change version number
085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 Remove unnecessary test
a11bef06a3f659402fe7563abf99ad00de2209e6 Initial commit
```

The most interesting option value is `format`, which allows you to specify your own log output format. This is especially useful when you're generating output for machine parsing — because you specify the format explicitly, you know it won't change with updates to Git:

```
$ git log --pretty=format:"%h - %an, %ar : %s"
ca82a6d - Scott Chacon, 6 years ago : Change version number
085bb3b - Scott Chacon, 6 years ago : Remove unnecessary test
a11bef0 - Scott Chacon, 6 years ago : Initial commit
```

Useful specifiers for `git log --pretty=format` lists some of the more useful specifiers that `format` takes.

Table 1. Useful specifiers for `git log --pretty=format`

Specifier	Description of Output
`%H`	Commit hash
`%h`	Abbreviated commit hash
`%T`	Tree hash
`%t`	Abbreviated tree hash
`%P`	Parent hashes
`%p`	Abbreviated parent hashes
`%an`	Author name
`%ae`	Author email
`%ad`	Author date (format respects the `--date=option`)
`%ar`	Author date, relative
`%cn`	Committer name
`%ce`	Committer email
`%cd`	Committer date
`%cr`	Committer date, relative
`%s`	Subject

You may be wondering what the difference is between *author* and *committer*. The author is the person who originally wrote the work, whereas the committer is the person who last applied the work. So, if you send in 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. We'll cover this distinction a bit more in Distributed Git.

The `oneline` and `format` option values are particularly useful with another `log` option called `--graph`. This option adds a nice little ASCII graph showing your branch and merge history:

```
$ git log --pretty=format:"%h %s" --graph
* 2d3acf9 Ignore errors from SIGCHLD on trap
*  5e3ee11 Merge branch 'master' of https://github.com/dustin/grit.git
|\
| * 420eac9 Add method for getting the current branch
* | 30e367c Timeout code and tests
```

```
* | 5a09431 Add timeout protection to grit
* | e1193f8 Support for heads with slashes in them
|/
* d6016bc Require time for xmlschema
*  11d191e Merge branch 'defunkt' into local
```

This type of output will become more interesting as we go through branching and merging in the next chapter.

Those are only some simple output-formatting options to `git log` — there are many more. Common options to `git log` lists the options we've covered so far, as well as some other common formatting options that may be useful, along with how they change the output of the `log` command.

Table 2. Common options to `git log`

Option	Description
`-p`	Show the patch introduced with each commit.
`--stat`	Show statistics for files modified in each commit.
`--shortstat`	Display only the changed/insertions/deletions line from the `--stat` command.
`--name-only`	Show the list of files modified after the commit information.
`--name-status`	Show the list of files affected with added/modified/deleted information as well.
`--abbrev-commit`	Show only the first few characters of the SHA-1 checksum instead of all 40.
`--relative-date`	Display the date in a relative format (for example, "2 weeks ago") instead of using the full date format.
`--graph`	Display an ASCII graph of the branch and merge history beside the log output.
`--pretty`	Show commits in an alternate format. Option values include `oneline`, `short`, `full`, `fuller`, and `format` (where you specify your own format).
`--oneline`	Shorthand for `--pretty=oneline --abbrev-commit` used together.

Limiting Log Output

In addition to output-formatting options, `git log` takes a number of useful limiting options; that is, options that let you show only a subset of commits. You've seen one such option already — the `-2` option, which displays only the last two commits. In fact, you can do `-<n>`, where `n` is any integer to show the last `n` commits. In reality, you're unlikely to use that often, because Git by default pipes all output through a pager so you see only one page of log output at a time.

However, the time-limiting options such as `--since` and `--until` are very useful. For example, this command gets the list of commits made in the last two weeks:

```
$ git log --since=2.weeks
```

This command works with lots of formats — you can specify a specific date like `"2008-01-15"`, or a relative date such as `"2 years 1 day 3 minutes ago"`.

You can also filter the list to commits that match some search criteria. The `--author` option allows you to filter on a specific author, and the `--grep` option lets you search for keywords in the commit messages.

You can specify more than one instance of both the `--author` and `--grep` search criteria, which will limit the commit output to commits that match *any* of the `--author` patterns and *any* of the `--grep` patterns; however, adding the `--all-match` option further limits the output to just those commits that match *all* `--grep` patterns.

Another really helpful filter is the `-S` option (colloquially referred to as Git's "pickaxe" option), which takes a string and shows only those commits that changed the number of occurrences of that string. For instance, if you wanted to find the last commit that added or removed a reference to a specific function, you could call:

```
$ git log -S function_name
```

The last really useful option to pass to `git log` as a filter is a path. If you specify a directory or file name, you can limit the log output to commits that introduced a change to those files. This is always the last option and is generally preceded by double dashes (`--`) to separate the paths from the options:

```
$ git log -- path/to/file
```

In Options to limit the output of `git log` we'll list these and a few other common options for your reference.

Table 3. Options to limit the output of `git log`

Option	Description
`-<n>`	Show only the last n commits.
`--since`, `--after`	Limit the commits to those made after the specified date.
`--until`, `--before`	Limit the commits to those made before the specified date.
`--author`	Only show commits in which the author entry matches the specified string.
`--committer`	Only show commits in which the committer entry matches the specified string.
`--grep`	Only show commits with a commit message containing the string.
`-S`	Only show commits adding or removing code matching the string.

For example, if you want to see which commits modifying test files in the Git source code history were committed by Junio Hamano in the month of October 2008 and are not merge commits, you

can run something like this:

```
$ git log --pretty="%h - %s" --author='Junio C Hamano' --since="2008-10-01" \
   --before="2008-11-01" --no-merges -- t/
5610e3b - Fix testcase failure when extended attributes are in use
acd3b9e - Enhance hold_lock_file_for_{update,append}() API
f563754 - demonstrate breakage of detached checkout with symbolic link HEAD
d1a43f2 - reset --hard/read-tree --reset -u: remove unmerged new paths
51a94af - Fix "checkout --track -b newbranch" on detached HEAD
b0ad11e - pull: allow "git pull origin $something:$current_branch" into an unborn
branch
```

Of the nearly 40,000 commits in the Git source code history, this command shows the 6 that match those criteria.

Preventing the display of merge commits

Depending on the workflow used in your repository, it's possible that a sizable percentage of the commits in your log history are just merge commits, which typically aren't very informative. To prevent the display of merge commits cluttering up your log history, simply add the `log` option `--no-merges`.

Undoing Things

At any stage, you may want to undo something. Here, we'll review a few basic tools for undoing changes that you've made. Be careful, because you can't always undo some of these undos. This is one of the few areas in Git where you may lose some work if you do it wrong.

One of the common undos takes place when you commit too early and possibly forget to add some files, or you mess up your commit message. If you want to redo that commit, make the additional changes you forgot, stage them, and commit again using the `--amend` option:

```
$ git commit --amend
```

This command takes your staging area and uses it for the commit. If you've made no changes since your last commit (for instance, you run this command immediately after your previous commit), then your snapshot will look exactly the same, and all you'll change is your commit message.

The same commit-message editor fires up, but it already contains the message of your previous commit. You can edit the message the same as always, but it overwrites your previous commit.

As an example, if you commit and then realize you forgot to stage the changes in a file you wanted to add to this commit, you can do something like this:

```
$ git commit -m 'Initial commit'
$ git add forgotten_file
```

```
$ git commit --amend
```

You end up with a single commit — the second commit replaces the results of the first.

It's important to understand that when you're amending your last commit, you're not so much fixing it as *replacing* it entirely with a new, improved commit that pushes the old commit out of the way and puts the new commit in its place. Effectively, it's as if the previous commit never happened, and it won't show up in your repository history.

The obvious value to amending commits is to make minor improvements to your last commit, without cluttering your repository history with commit messages of the form, "Oops, forgot to add a file" or "Darn, fixing a typo in last commit".

Only amend commits that are still local and have not been pushed somewhere. Amending previously pushed commits and force pushing the branch will cause problems for your collaborators. For more on what happens when you do this and how to recover if you're on the receiving end read The Perils of Rebasing.

Unstaging a Staged File

The next two sections demonstrate how to work with your staging area and working directory changes. The nice part is that the command you use to determine the state of those two areas also reminds you how to undo changes to them. For example, let's say you've changed two files and want to commit them as two separate changes, but you accidentally type `git add *` and stage them both. How can you unstage one of the two? The `git status` command reminds you:

```
$ git add *
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed:    README.md -> README
    modified:   CONTRIBUTING.md
```

Right below the "Changes to be committed" text, it says use `git reset HEAD <file>⋯` to unstage. So, let's use that advice to unstage the `CONTRIBUTING.md` file:

```
$ git reset HEAD CONTRIBUTING.md
Unstaged changes after reset:
M   CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
```

```
    renamed:    README.md -> README

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:   CONTRIBUTING.md
```

The command is a bit strange, but it works. The `CONTRIBUTING.md` file is modified but once again unstaged.

It's true that `git reset` can be a dangerous command, especially if you provide the `--hard` flag. However, in the scenario described above, the file in your working directory is not touched, so it's relatively safe.

For now this magic invocation is all you need to know about the `git reset` command. We'll go into much more detail about what `reset` does and how to master it to do really interesting things in Reset Demystified.

Unmodifying a Modified File

What if you realize that you don't want to keep your changes to the `CONTRIBUTING.md` file? How can you easily unmodify it — revert it back to what it looked like when you last committed (or initially cloned, or however you got it into your working directory)? Luckily, `git status` tells you how to do that, too. In the last example output, the unstaged area looks like this:

```
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:   CONTRIBUTING.md
```

It tells you pretty explicitly how to discard the changes you've made. Let's do what it says:

```
$ git checkout -- CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed:    README.md -> README
```

You can see that the changes have been reverted.

It's important to understand that `git checkout -- <file>` is a dangerous command. Any local changes you made to that file are gone — Git just replaced that file with the last staged or committed version. Don't ever use this command unless you

> absolutely know that you don't want those unsaved local changes.

If you would like to keep the changes you've made to that file but still need to get it out of the way for now, we'll go over stashing and branching in Git Branching; these are generally better ways to go.

Remember, anything that is *committed* in Git can almost always be recovered. Even commits that were on branches that were deleted or commits that were overwritten with an `--amend` commit can be recovered (see Data Recovery for data recovery). However, anything you lose that was never committed is likely never to be seen again.

Undoing things with git restore

Git version 2.23.0 introduced a new command: `git restore`. It's basically an alternative to `git reset` which we just covered. From Git version 2.23.0 onwards, Git will use `git restore` instead of `git reset` for many undo operations.

Let's retrace our steps, and undo things with `git restore` instead of `git reset`.

Unstaging a Staged File with git restore

The next two sections demonstrate how to work with your staging area and working directory changes with `git restore`. The nice part is that the command you use to determine the state of those two areas also reminds you how to undo changes to them. For example, let's say you've changed two files and want to commit them as two separate changes, but you accidentally type `git add *` and stage them both. How can you unstage one of the two? The `git status` command reminds you:

```
$ git add *
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   CONTRIBUTING.md
    renamed:    README.md -> README
```

Right below the "Changes to be committed" text, it says use `git restore --staged <file>…` to unstage. So, let's use that advice to unstage the `CONTRIBUTING.md` file:

```
$ git restore --staged CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    renamed:    README.md -> README

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
```

```
    modified:   CONTRIBUTING.md
```

The `CONTRIBUTING.md` file is modified but once again unstaged.

Unmodifying a Modified File with git restore

What if you realize that you don't want to keep your changes to the `CONTRIBUTING.md` file? How can you easily unmodify it — revert it back to what it looked like when you last committed (or initially cloned, or however you got it into your working directory)? Luckily, `git status` tells you how to do that, too. In the last example output, the unstaged area looks like this:

```
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   CONTRIBUTING.md
```

It tells you pretty explicitly how to discard the changes you've made. Let's do what it says:

```
$ git restore CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    renamed:    README.md -> README
```

It's important to understand that `git restore <file>` is a dangerous command. Any local changes you made to that file are gone — Git just replaced that file with the last staged or committed version. Don't ever use this command unless you absolutely know that you don't want those unsaved local changes.

Working with Remotes

To be able to collaborate on any Git project, you need to know how to manage your remote repositories. Remote repositories are versions of your project that are hosted on the Internet or network somewhere. You can have several of them, each of which generally is either read-only or read/write for you. Collaborating with others involves managing these remote repositories and pushing and pulling data to and from them when you need to share work. Managing remote repositories includes knowing how to add remote repositories, remove remotes that are no longer valid, manage various remote branches and define them as being tracked or not, and more. In this section, we'll cover some of these remote-management skills.

Remote repositories can be on your local machine.

It is entirely possible that you can be working with a "remote" repository that is, in fact, on the same host you are. The word "remote" does not necessarily imply that the repository is somewhere else on the network or Internet, only that it is

> elsewhere. Working with such a remote repository would still involve all the standard pushing, pulling and fetching operations as with any other remote.

Showing Your Remotes

To see which remote servers you have configured, you can run the `git remote` command. It lists the shortnames of each remote handle you've specified. If you've cloned your repository, you should at least see `origin` — that is the default name Git gives to the server you cloned from:

```
$ git clone https://github.com/schacon/ticgit
Cloning into 'ticgit'...
remote: Reusing existing pack: 1857, done.
remote: Total 1857 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (1857/1857), 374.35 KiB | 268.00 KiB/s, done.
Resolving deltas: 100% (772/772), done.
Checking connectivity... done.
$ cd ticgit
$ git remote
origin
```

You can also specify `-v`, which shows you the URLs that Git has stored for the shortname to be used when reading and writing to that remote:

```
$ git remote -v
origin  https://github.com/schacon/ticgit (fetch)
origin  https://github.com/schacon/ticgit (push)
```

If you have more than one remote, the command lists them all. For example, a repository with multiple remotes for working with several collaborators might look something like this.

```
$ cd grit
$ git remote -v
bakkdoor  https://github.com/bakkdoor/grit (fetch)
bakkdoor  https://github.com/bakkdoor/grit (push)
cho45     https://github.com/cho45/grit (fetch)
cho45     https://github.com/cho45/grit (push)
defunkt   https://github.com/defunkt/grit (fetch)
defunkt   https://github.com/defunkt/grit (push)
koke      git://github.com/koke/grit.git (fetch)
koke      git://github.com/koke/grit.git (push)
origin    git@github.com:mojombo/grit.git (fetch)
origin    git@github.com:mojombo/grit.git (push)
```

This means we can pull contributions from any of these users pretty easily. We may additionally have permission to push to one or more of these, though we can't tell that here.

Notice that these remotes use a variety of protocols; we'll cover more about this in Getting Git on a

Server.

Adding Remote Repositories

We've mentioned and given some demonstrations of how the `git clone` command implicitly adds the `origin` remote for you. Here's how to add a new remote explicitly. To add a new remote Git repository as a shortname you can reference easily, run `git remote add <shortname> <url>`:

```
$ git remote
origin
$ git remote add pb https://github.com/paulboone/ticgit
$ git remote -v
origin  https://github.com/schacon/ticgit (fetch)
origin  https://github.com/schacon/ticgit (push)
pb  https://github.com/paulboone/ticgit (fetch)
pb  https://github.com/paulboone/ticgit (push)
```

Now you can use the string `pb` on the command line instead of the whole URL. For example, if you want to fetch all the information that Paul has but that you don't yet have in your repository, you can run `git fetch pb`:

```
$ git fetch pb
remote: Counting objects: 43, done.
remote: Compressing objects: 100% (36/36), done.
remote: Total 43 (delta 10), reused 31 (delta 5)
Unpacking objects: 100% (43/43), done.
From https://github.com/paulboone/ticgit
 * [new branch]      master     -> pb/master
 * [new branch]      ticgit     -> pb/ticgit
```

Paul's `master` branch is now accessible locally as `pb/master` — you can merge it into one of your branches, or you can check out a local branch at that point if you want to inspect it. We'll go over what branches are and how to use them in much more detail in Git Branching.

Fetching and Pulling from Your Remotes

As you just saw, to get data from your remote projects, you can run:

```
$ git fetch <remote>
```

The command goes out to that remote project and pulls down all the data from that remote project that you don't have yet. After you do this, you should have references to all the branches from that remote, which you can merge in or inspect at any time.

If you clone a repository, the command automatically adds that remote repository under the name "origin". So, `git fetch origin` fetches any new work that has been pushed to that server since you

cloned (or last fetched from) it. It's important to note that the `git fetch` command only downloads the data to your local repository — it doesn't automatically merge it with any of your work or modify what you're currently working on. You have to merge it manually into your work when you're ready.

If your current branch is set up to track a remote branch (see the next section and Git Branching for more information), you can use the `git pull` command to automatically fetch and then merge that remote branch into your current branch. This may be an easier or more comfortable workflow for you; and by default, the `git clone` command automatically sets up your local `master` branch to track the remote `master` branch (or whatever the default branch is called) on the server you cloned from. Running `git pull` generally fetches data from the server you originally cloned from and automatically tries to merge it into the code you're currently working on.

From Git version 2.27 onward, `git pull` will give a warning if the `pull.rebase` variable is not set. Git will keep warning you until you set the variable.

If you want the default behavior of Git (fast-forward if possible, else create a merge commit): `git config --global pull.rebase "false"`

If you want to rebase when pulling: `git config --global pull.rebase "true"`

Pushing to Your Remotes

When you have your project at a point that you want to share, you have to push it upstream. The command for this is simple: `git push <remote> <branch>`. If you want to push your `master` branch to your `origin` server (again, cloning generally sets up both of those names for you automatically), then you can run this to push any commits you've done back up to the server:

```
$ git push origin master
```

This command works only if you cloned from a server to which you have write access and if nobody has pushed in the meantime. If you and someone else clone at the same time and they push upstream and then you push upstream, your push will rightly be rejected. You'll have to fetch their work first and incorporate it into yours before you'll be allowed to push. See Git Branching for more detailed information on how to push to remote servers.

Inspecting a Remote

If you want to see more information about a particular remote, you can use the `git remote show <remote>` command. If you run this command with a particular shortname, such as `origin`, you get something like this:

```
$ git remote show origin
* remote origin
  Fetch URL: https://github.com/schacon/ticgit
  Push  URL: https://github.com/schacon/ticgit
  HEAD branch: master
```

```
  Remote branches:
    master                               tracked
    dev-branch                           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)
```

It lists the URL for the remote repository as well as the tracking branch information. The command helpfully tells you that if you're on the `master` branch and you run `git pull`, it will automatically merge the remote's `master` branch into the local one after it has been fetched. It also lists all the remote references it has pulled down.

That is a simple example you're likely to encounter. When you're using Git more heavily, however, you may see much more information from `git remote show`:

```
$ git remote show origin
* remote origin
  URL: https://github.com/my-org/complex-project
  Fetch URL: https://github.com/my-org/complex-project
  Push  URL: https://github.com/my-org/complex-project
  HEAD branch: master
  Remote branches:
    master                           tracked
    dev-branch                       tracked
    markdown-strip                   tracked
    issue-43                         new (next fetch will store in remotes/origin)
    issue-45                         new (next fetch will store in remotes/origin)
    refs/remotes/origin/issue-11     stale (use 'git remote prune' to remove)
  Local branches configured for 'git pull':
    dev-branch merges with remote dev-branch
    master     merges with remote master
  Local refs configured for 'git push':
    dev-branch                     pushes to dev-branch                     (up to
date)
    markdown-strip                 pushes to markdown-strip                 (up to
date)
    master                         pushes to master                         (up to
date)
```

This command shows which branch is automatically pushed to when you run `git push` while on certain branches. It also shows you which remote branches on the server you don't yet have, which remote branches you have that have been removed from the server, and multiple local branches that are able to merge automatically with their remote-tracking branch when you run `git pull`.

Renaming and Removing Remotes

You can run `git remote rename` to change a remote's shortname. For instance, if you want to rename `pb` to `paul`, you can do so with `git remote rename`:

```
$ git remote rename pb paul
$ git remote
origin
paul
```

It's worth mentioning that this changes all your remote-tracking branch names, too. What used to be referenced at `pb/master` is now at `paul/master`.

If you want to remove a remote for some reason — you've moved the server or are no longer using a particular mirror, or perhaps a contributor isn't contributing anymore — you can either use `git remote remove` or `git remote rm`:

```
$ git remote remove paul
$ git remote
origin
```

Once you delete the reference to a remote this way, all remote-tracking branches and configuration settings associated with that remote are also deleted.

Tagging

Like most VCSs, Git has the ability to tag specific points in a repository's history as being important. Typically, people use this functionality to mark release points (`v1.0`, `v2.0` and so on). In this section, you'll learn how to list existing tags, how to create and delete tags, and what the different types of tags are.

Listing Your Tags

Listing the existing tags in Git is straightforward. Just type `git tag` (with optional `-l` or `--list`):

```
$ git tag
v1.0
v2.0
```

This command lists the tags in alphabetical order; the order in which they are displayed has no real importance.

You can also search for tags that match a particular pattern. The Git source repo, for instance, contains more than 500 tags. If you're interested only in looking at the 1.8.5 series, you can run this:

```
$ git tag -l "v1.8.5*"
v1.8.5
v1.8.5-rc0
v1.8.5-rc1
v1.8.5-rc2
v1.8.5-rc3
```

```
v1.8.5.1
v1.8.5.2
v1.8.5.3
v1.8.5.4
v1.8.5.5
```

Listing tag wildcards requires `-l` or `--list` option

If you want just the entire list of tags, running the command `git tag` implicitly assumes you want a listing and provides one; the use of `-l` or `--list` in this case is optional.

If, however, you're supplying a wildcard pattern to match tag names, the use of `-l` or `--list` is mandatory.

Creating Tags

Git supports 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 to a specific commit.

Annotated tags, however, are stored as full objects in the Git database. They're checksummed; contain the tagger name, email, and date; have a tagging message; and can be signed and verified with GNU Privacy Guard (GPG). It's generally recommended that you create annotated tags so you can have all this information; but if you want a temporary tag or for some reason don't want to keep the other information, lightweight tags are available too.

Annotated Tags

Creating an annotated tag in Git is simple. The easiest way is to specify `-a` when you run the `tag` command:

```
$ git tag -a v1.4 -m "my version 1.4"
$ git tag
v0.1
v1.3
v1.4
```

The `-m` specifies a tagging message, which is stored with the tag. If you don't specify a message for an annotated tag, Git launches your editor so you can type it in.

You can see the tag data along with the commit that was tagged by using the `git show` command:

```
$ git show v1.4
tag v1.4
Tagger: Ben Straub <ben@straub.cc>
Date:   Sat May 3 20:19:12 2014 -0700
```

```
my version 1.4

commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    Change version number
```

That shows the tagger information, the date the commit was tagged, and the annotation message before showing the commit information.

Lightweight Tags

Another way to tag commits is with a lightweight tag. This is basically the commit checksum stored in a file — no other information is kept. To create a lightweight tag, don't supply any of the `-a`, `-s`, or `-m` options, just provide a tag name:

```
$ git tag v1.4-lw
$ git tag
v0.1
v1.3
v1.4
v1.4-lw
v1.5
```

This time, if you run `git show` on the tag, you don't see the extra tag information. The command just shows the commit:

```
$ git show v1.4-lw
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    Change version number
```

Tagging Later

You can also tag commits after you've moved past them. Suppose your commit history looks like this:

```
$ git log --pretty=oneline
15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'
a6b4c97498bd301d84096da251c98a07c7723e65 Create write support
0d52aaab4479697da7686c15f77a3d64d9165190 One more thing
6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'
0b7434d86859cc7b8c3d5e1dddfed66ff742fcbc Add commit function
```

```
4682c3261057305bdd616e23b64b0857d832627b Add todo file
166ae0c4d3f420721acbb115cc33848dfcc2121a Create write support
9fceb02d0ae598e95dc970b74767f19372d61af8 Update rakefile
964f16d36dfccde844893cac5b347e7b3d44abbc Commit the todo
8a5cbc430f1a9c3d00faaeffd07798508422908a Update readme
```

Now, suppose you forgot to tag the project at v1.2, which was at the “Update rakefile” commit. You can add it after the fact. To tag that commit, you specify the commit checksum (or part of it) at the end of the command:

```
$ git tag -a v1.2 9fceb02
```

You can see that you’ve tagged the commit:

```
$ git tag
v0.1
v1.2
v1.3
v1.4
v1.4-lw
v1.5

$ git show v1.2
tag v1.2
Tagger: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Feb 9 15:32:16 2009 -0800

version 1.2
commit 9fceb02d0ae598e95dc970b74767f19372d61af8
Author: Magnus Chacon <mchacon@gee-mail.com>
Date:   Sun Apr 27 20:43:35 2008 -0700

    Update rakefile
...
```

Sharing Tags

By default, the `git push` command doesn’t transfer tags to remote servers. You will have to explicitly push tags to a shared server after you have created them. This process is just like sharing remote branches — you can run `git push origin <tagname>`.

```
$ git push origin v1.5
Counting objects: 14, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (12/12), done.
Writing objects: 100% (14/14), 2.05 KiB | 0 bytes/s, done.
Total 14 (delta 3), reused 0 (delta 0)
```

```
To git@github.com:schacon/simplegit.git
 * [new tag]         v1.5 -> v1.5
```

If you have a lot of tags that you want to push up at once, you can also use the `--tags` option to the `git push` command. This will transfer all of your tags to the remote server that are not already there.

```
$ git push origin --tags
Counting objects: 1, done.
Writing objects: 100% (1/1), 160 bytes | 0 bytes/s, done.
Total 1 (delta 0), reused 0 (delta 0)
To git@github.com:schacon/simplegit.git
 * [new tag]         v1.4 -> v1.4
 * [new tag]         v1.4-lw -> v1.4-lw
```

Now, when someone else clones or pulls from your repository, they will get all your tags as well.

`git push` *pushes both types of tags*

`git push <remote> --tags` will push both lightweight and annotated tags. There is currently no option to push only lightweight tags, but if you use `git push <remote> --follow-tags` only annotated tags will be pushed to the remote.

Deleting Tags

To delete a tag on your local repository, you can use `git tag -d <tagname>`. For example, we could remove our lightweight tag above as follows:

```
$ git tag -d v1.4-lw
Deleted tag 'v1.4-lw' (was e7d5add)
```

Note that this does not remove the tag from any remote servers. There are two common variations for deleting a tag from a remote server.

The first variation is `git push <remote> :refs/tags/<tagname>`:

```
$ git push origin :refs/tags/v1.4-lw
To /git@github.com:schacon/simplegit.git
 - [deleted]         v1.4-lw
```

The way to interpret the above is to read it as the null value before the colon is being pushed to the remote tag name, effectively deleting it.

The second (and more intuitive) way to delete a remote tag is with:

```
$ git push origin --delete <tagname>
```

Checking out Tags

If you want to view the versions of files a tag is pointing to, you can do a `git checkout` of that tag, although this puts your repository in "detached HEAD" state, which has some ill side effects:

```
$ git checkout v2.0.0
Note: switching to 'v2.0.0'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -c with the switch command. Example:

  git switch -c <new-branch-name>

Or undo this operation with:

  git switch -

Turn off this advice by setting config variable advice.detachedHead to false

HEAD is now at 99ada87... Merge pull request #89 from schacon/appendix-final

$ git checkout v2.0-beta-0.1
Previous HEAD position was 99ada87... Merge pull request #89 from schacon/appendix-
final
HEAD is now at df3f601... Add atlas.json and cover image
```

In "detached HEAD" state, if you make changes and then create a commit, the tag will stay the same, but your new commit won't belong to any branch and will be unreachable, except by the exact commit hash. Thus, if you need to make changes — say you're fixing a bug on an older version, for instance — you will generally want to create a branch:

```
$ git checkout -b version2 v2.0.0
Switched to a new branch 'version2'
```

If you do this and make a commit, your `version2` branch will be slightly different than your `v2.0.0` tag since it will move forward with your new changes, so do be careful.

Git Aliases

Before we move on to the next chapter, we want to introduce a feature that can make your Git

experience simpler, easier, and more familiar: aliases. For clarity's sake, we won't be using them anywhere else in this book, but if you go on to use Git with any regularity, aliases are something you should know about.

Git doesn't automatically infer your command if you type it in partially. If you don't want to type the entire text of each of the Git commands, you can easily set up an alias for each command using `git config`. Here are a couple of examples you may want to set up:

```
$ git config --global alias.co checkout
$ git config --global alias.br branch
$ git config --global alias.ci commit
$ git config --global alias.st status
```

This means that, for example, instead of typing `git commit`, you just need to type `git ci`. As you go on using Git, you'll probably use other commands frequently as well; don't hesitate to create new aliases.

This technique can also be very useful in creating commands that you think should exist. For example, to correct the usability problem you encountered with unstaging a file, you can add your own unstage alias to Git:

```
$ git config --global alias.unstage 'reset HEAD --'
```

This makes the following two commands equivalent:

```
$ git unstage fileA
$ git reset HEAD -- fileA
```

This seems a bit clearer. It's also common to add a `last` command, like this:

```
$ git config --global alias.last 'log -1 HEAD'
```

This way, you can see the last commit easily:

```
$ git last
commit 66938dae3329c7aebe598c2246a8e6af90d04646
Author: Josh Goebel <dreamer3@example.com>
Date:   Tue Aug 26 19:48:51 2008 +0800

    Test for current head

    Signed-off-by: Scott Chacon <schacon@example.com>
```

As you can tell, Git simply replaces the new command with whatever you alias it for. However, maybe you want to run an external command, rather than a Git subcommand. In that case, you

start the command with a `!` character. This is useful if you write your own tools that work with a Git repository. We can demonstrate by aliasing `git visual` to run `gitk`:

```
$ git config --global alias.visual '!gitk'
```

Summary

At this point, you can do all the basic local Git operations — creating or cloning a repository, making changes, staging and committing those changes, and viewing the history of all the changes the repository has been through. Next, we'll cover Git's killer feature: its branching model.

Git Branching

Nearly every VCS has some form of branching support. Branching means you diverge from the main line of development and continue to do work without messing with that main line. In many VCS tools, this is a somewhat expensive process, often requiring you to create a new copy of your source code directory, which can take a long time for large projects.

Some people refer to Git's branching model as its "killer feature," and it certainly sets Git apart in the VCS community. Why is it so special? The way Git branches is incredibly lightweight, making branching operations nearly instantaneous, and switching back and forth between branches generally just as fast. Unlike many other VCSs, Git encourages workflows that branch and merge often, even multiple times in a day. Understanding and mastering this feature gives you a powerful and unique tool and can entirely change the way that you develop.

Branches in a Nutshell

To really understand the way Git does branching, we need to take a step back and examine how Git stores its data.

As you may remember from What is Git?, Git doesn't store data as a series of changesets or differences, but instead as a series of *snapshots*.

When you make a commit, Git stores a commit object that contains a pointer to the snapshot of the content you staged. This object also contains the author's name and email address, the message that you typed, and pointers to the commit or commits that directly came before this commit (its parent or parents): zero parents for the initial commit, one parent for a normal commit, and multiple parents for a commit that results from a merge of two or more branches.

To visualize this, let's assume that you have a directory containing three files, and you stage them all and commit. Staging the files computes a checksum for each one (the SHA-1 hash we mentioned in What is Git?), stores that version of the file in the Git repository (Git refers to them as *blobs*), and adds that checksum to the staging area:

```
$ git add README test.rb LICENSE
$ git commit -m 'Initial commit'
```

When you create the commit by running `git commit`, Git checksums each subdirectory (in this case, just the root project directory) and stores them as a tree object in the Git repository. Git then creates a commit object that has the metadata and a pointer to the root project tree so it can re-create that snapshot when needed.

Your Git repository now contains five objects: three *blobs* (each representing the contents of one of the three files), one *tree* that lists the contents of the directory and specifies which file names are stored as which blobs, and one *commit* with the pointer to that root tree and all the commit metadata.

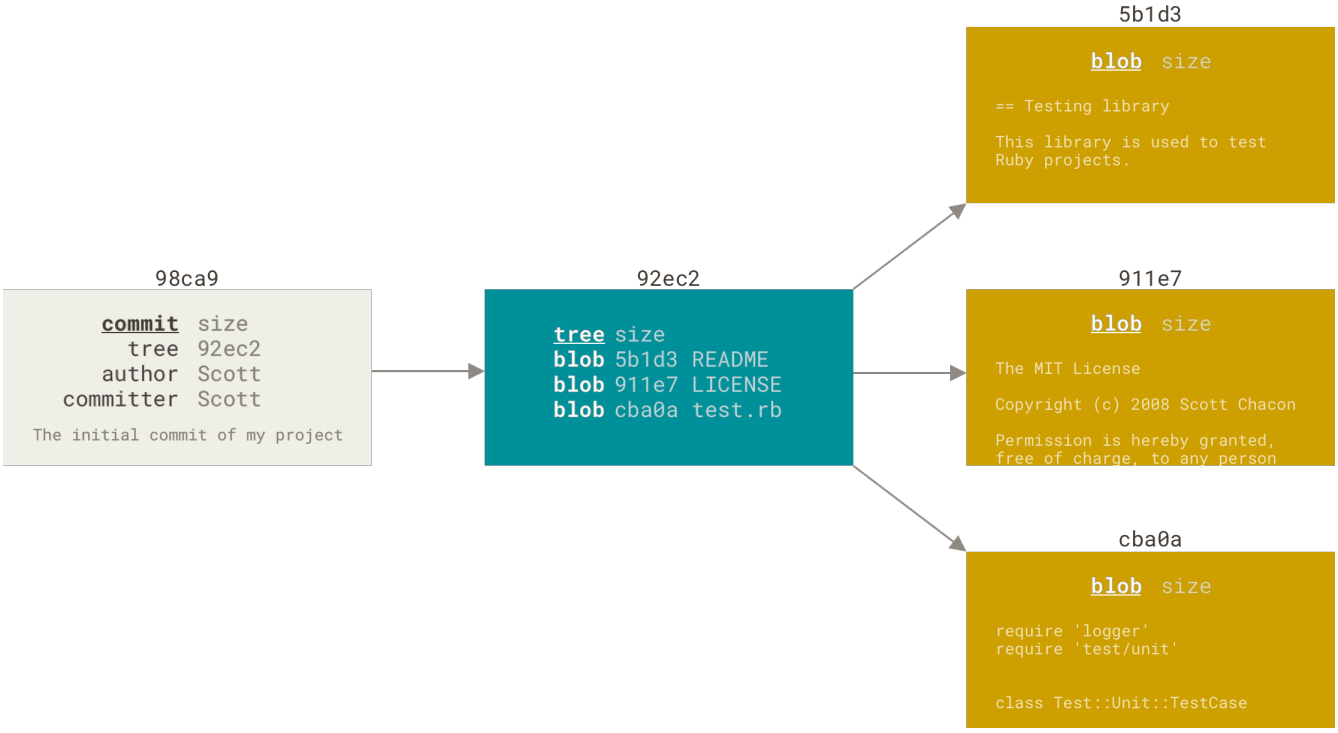

Figure 9. A commit and its tree

If you make some changes and commit again, the next commit stores a pointer to the commit that came immediately before it.

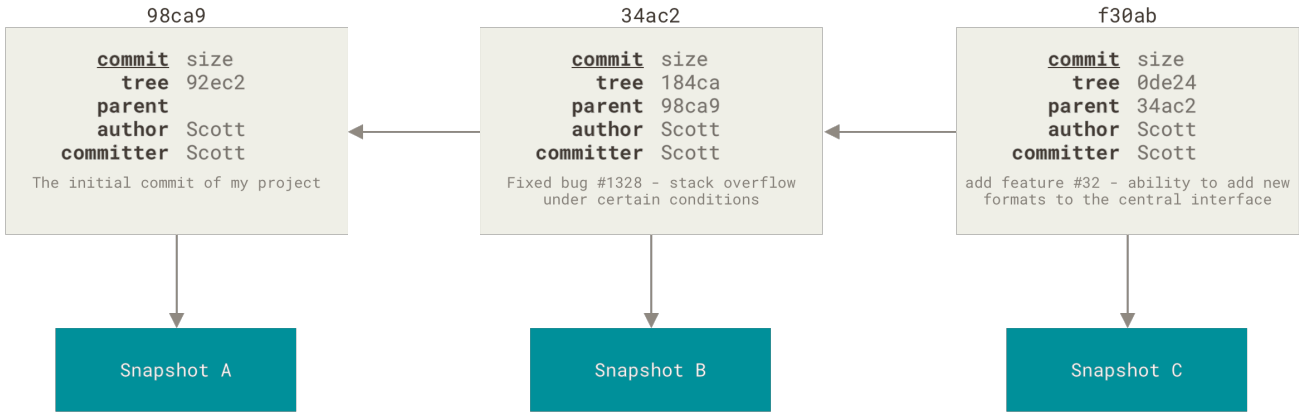

Figure 10. Commits and their parents

A branch in Git is simply a lightweight movable pointer to one of these commits. The default branch name in Git is `master`. As you start making commits, you're given a `master` branch that points to the last commit you made. Every time you commit, the `master` branch pointer moves forward automatically.

The "master" branch in Git is not a special branch. It is exactly like any other branch. The only reason nearly every repository has one is that the `git init` command creates it by default and most people don't bother to change it.

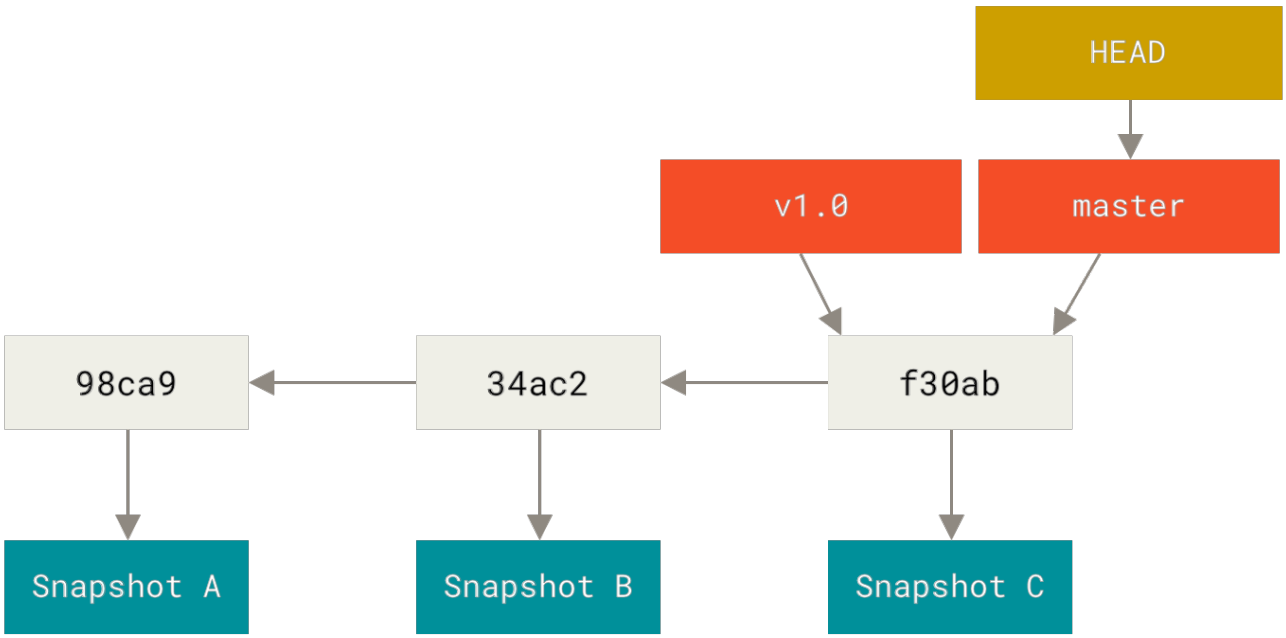

Figure 11. A branch and its commit history

Creating a New Branch

What happens when you create a new branch? Well, doing so creates a new pointer for you to move around. Let's say you want to create a new branch called `testing`. You do this with the `git branch` command:

```
$ git branch testing
```

This creates a new pointer to the same commit you're currently on.

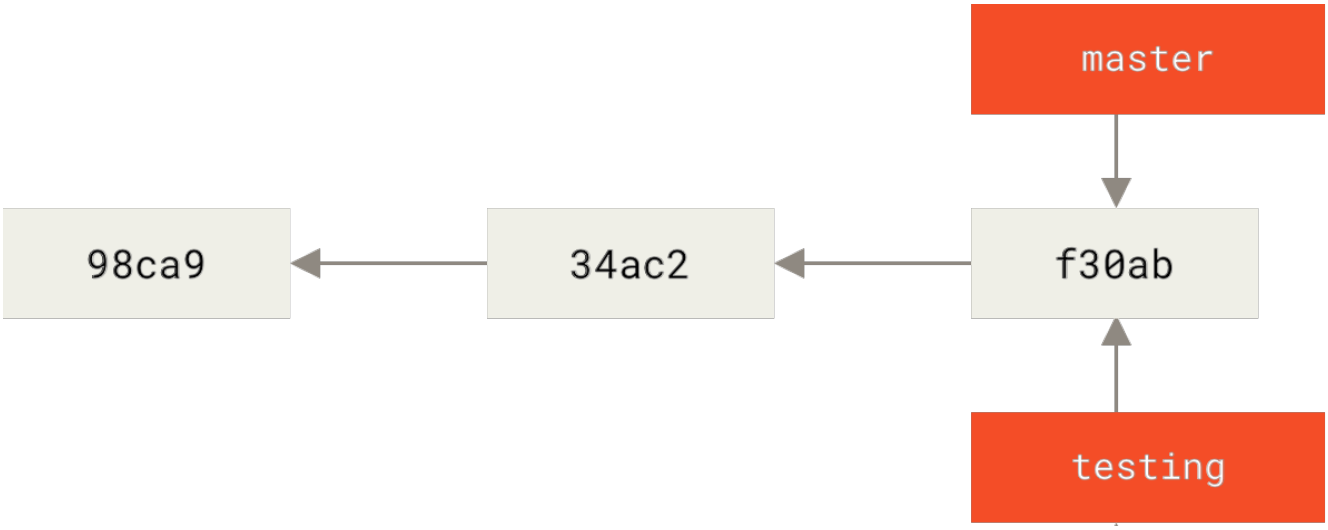

Figure 12. Two branches pointing into the same series of commits

How does Git know what branch you're currently on? It keeps a special pointer called `HEAD`. Note that this is a lot different than the concept of `HEAD` in other VCSs you may be used to, such as Subversion or CVS. In Git, this is a pointer to the local branch you're currently on. In this case, you're still on `master`. The `git branch` command only *created* a new branch — it didn't switch to that

branch.

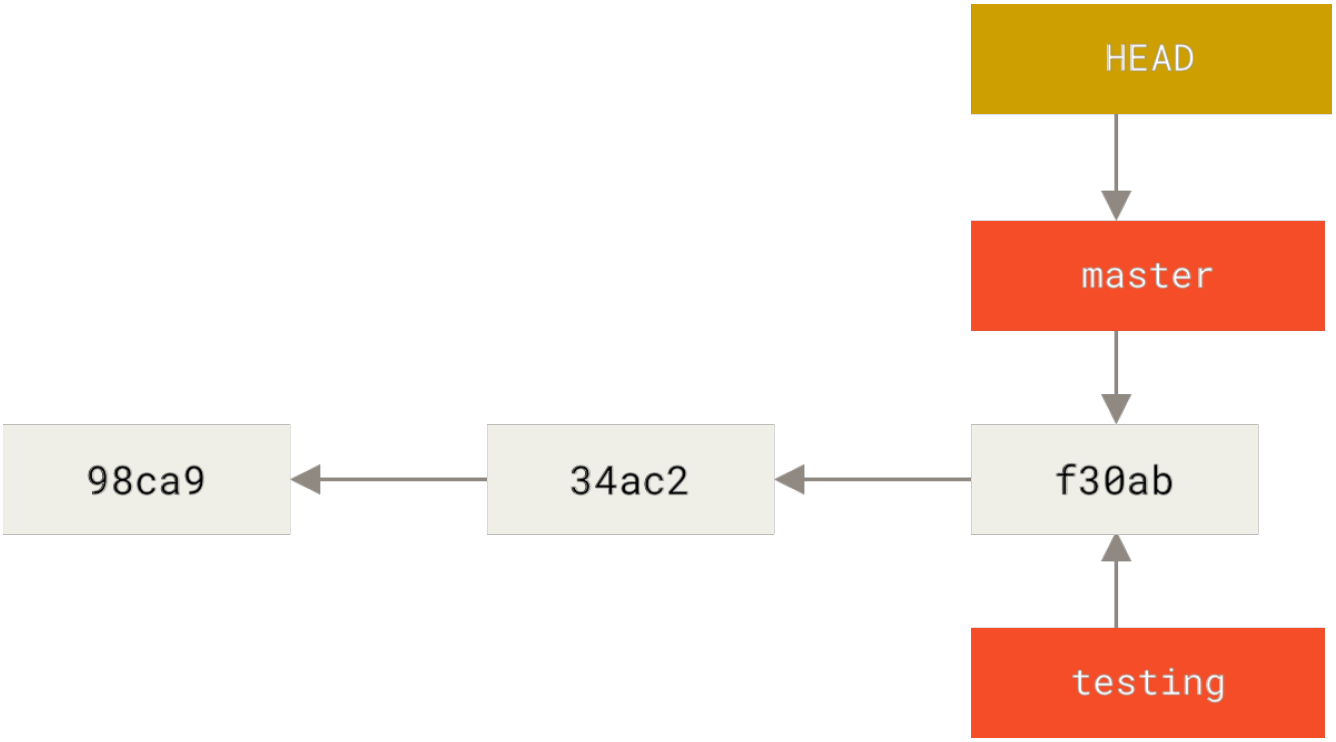

Figure 13. HEAD pointing to a branch

You can easily see this by running a simple `git log` command that shows you where the branch pointers are pointing. This option is called `--decorate`.

```
$ git log --oneline --decorate
f30ab (HEAD -> master, testing) Add feature #32 - ability to add new formats to the
central interface
34ac2 Fix bug #1328 - stack overflow under certain conditions
98ca9 Initial commit
```

You can see the `master` and `testing` branches that are right there next to the `f30ab` commit.

Switching Branches

To switch to an existing branch, you run the `git checkout` command. Let's switch to the new `testing` branch:

```
$ git checkout testing
```

This moves `HEAD` to point to the `testing` branch.

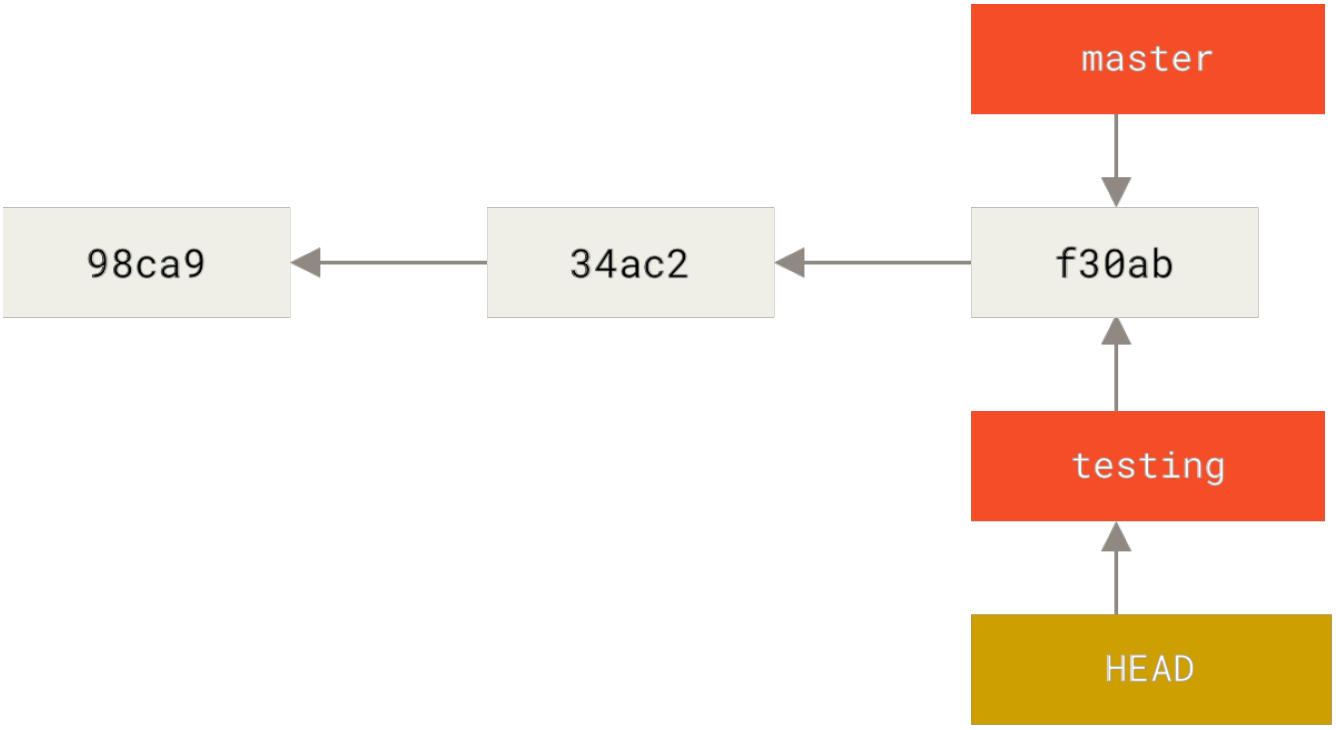

Figure 14. HEAD points to the current branch

What is the significance of that? Well, let's do another commit:

```
$ vim test.rb
$ git commit -a -m 'Make a change'
```

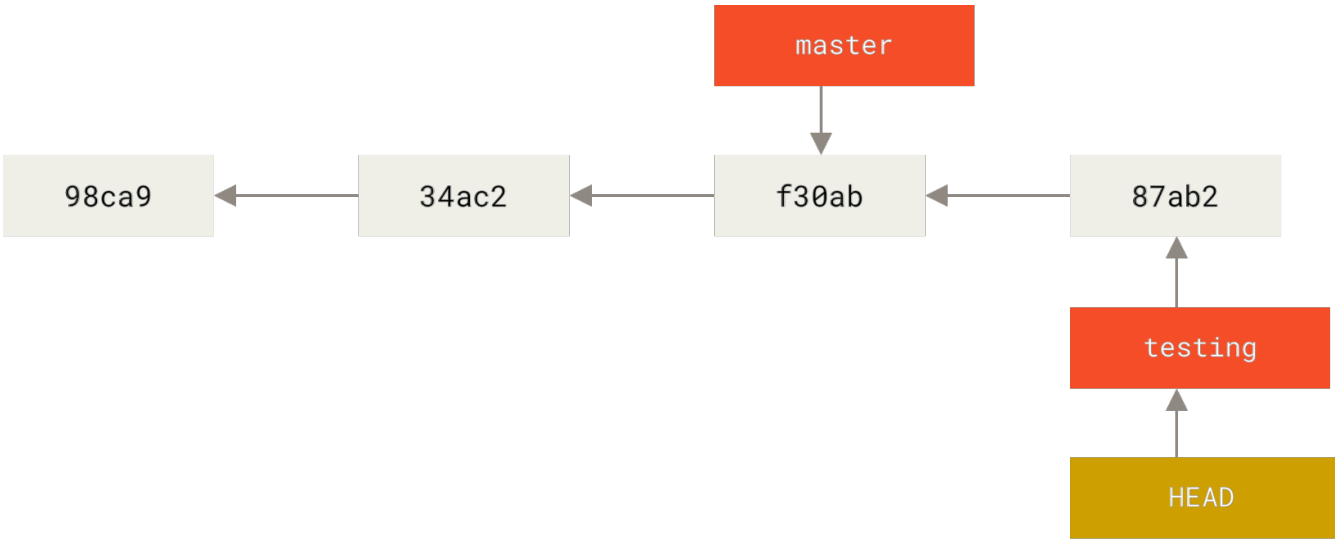

Figure 15. The HEAD branch moves forward when a commit is made

This is interesting, because now your `testing` branch has moved forward, but your `master` branch still points to the commit you were on when you ran `git checkout` to switch branches. Let's switch back to the `master` branch:

```
$ git checkout master
```

`git log` *doesn't show all the branches all the time*

If you were to run `git log` right now, you might wonder where the "testing" branch you just created went, as it would not appear in the output.

The branch hasn't disappeared; Git just doesn't know that you're interested in that branch and it is trying to show you what it thinks you're interested in. In other words, by default, `git log` will only show commit history below the branch you've checked out.

To show commit history for the desired branch you have to explicitly specify it: `git log testing`. To show all of the branches, add `--all` to your `git log` command.

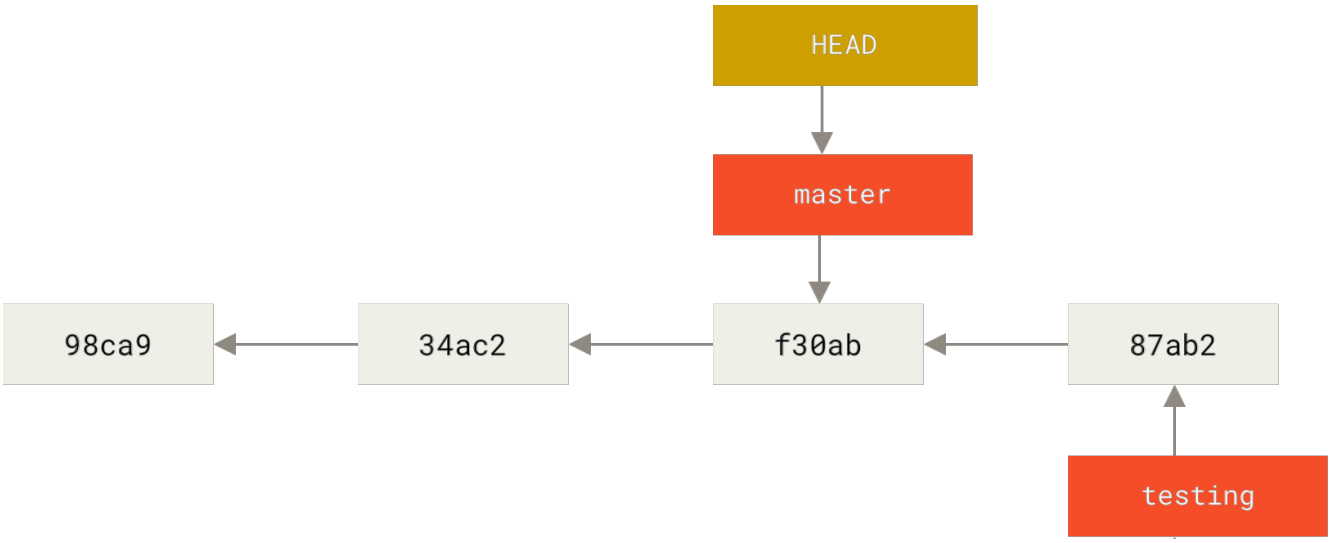

Figure 16. HEAD moves when you checkout

That command did two things. It moved the HEAD pointer back to point to the `master` branch, and it reverted the files in your working directory back to the snapshot that `master` points to. This also means the changes you make from this point forward will diverge from an older version of the project. It essentially rewinds the work you've done in your `testing` branch so you can go in a different direction.

Switching branches changes files in your working directory

It's important to note that when you switch branches in Git, files in your working directory will change. If you switch to an older branch, your working directory will be reverted to look like it did the last time you committed on that branch. If Git cannot do it cleanly, it will not let you switch at all.

Let's make a few changes and commit again:

```
$ vim test.rb
$ git commit -a -m 'Make other changes'
```

Now your project history has diverged (see Divergent history). You created and switched to a branch, did some work on it, and then switched back to your main branch and did other work. Both of those changes are isolated in separate branches: you can switch back and forth between the branches and merge them together when you're ready. And you did all that with simple `branch`,

`checkout`, and `commit` commands.

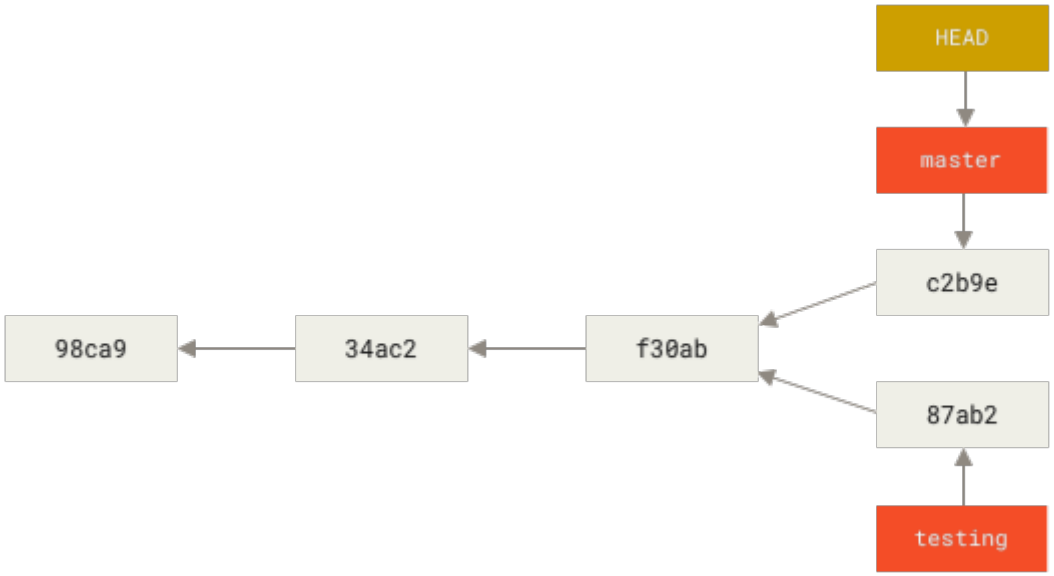

Figure 17. Divergent history

You can also see this easily with the `git log` command. If you run `git log --oneline --decorate --graph --all` it will print out the history of your commits, showing where your branch pointers are and how your history has diverged.

```
$ git log --oneline --decorate --graph --all
* c2b9e (HEAD, master) Make other changes
| * 87ab2 (testing) Make a change
|/
* f30ab Add feature #32 - ability to add new formats to the central interface
* 34ac2 Fix bug #1328 - stack overflow under certain conditions
* 98ca9 Initial commit of my project
```

Because a branch in Git is actually a simple file that contains the 40 character SHA-1 checksum of the commit it points to, branches are cheap to create and destroy. Creating a new branch is as quick and simple as writing 41 bytes to a file (40 characters and a newline).

This is in sharp contrast to the way most older VCS tools branch, which involves copying all of the project's files into a second directory. This can take several seconds or even minutes, depending on the size of the project, whereas in Git the process is always instantaneous. Also, because we're recording the parents when we commit, finding a proper merge base for merging is automatically done for us and is generally very easy to do. These features help encourage developers to create and use branches often.

Let's see why you should do so.

Creating a new branch and switching to it at the same time

It's typical to create a new branch and want to switch to that new branch at the same time — this can be done in one operation with `git checkout -b <newbranchname>`.

From Git version 2.23 onwards you can use `git switch` instead of `git checkout` to:

- Switch to an existing branch: `git switch testing-branch`.
- Create a new branch and switch to it: `git switch -c new-branch`. The `-c` flag stands for create, you can also use the full flag: `--create`.
- Return to your previously checked out branch: `git switch -`.

Basic Branching and Merging

Let's go through a simple example of branching and merging with a workflow that you might use in the real world. You'll follow these steps:

1. Do some work on a website.
2. Create a branch for a new user story you're working on.
3. Do some work in that branch.

At this stage, you'll receive a call that another issue is critical and you need a hotfix. You'll do the following:

1. Switch to your production branch.
2. Create a branch to add the hotfix.
3. After it's tested, merge the hotfix branch, and push to production.
4. Switch back to your original user story and continue working.

Basic Branching

First, let's say you're working on your project and have a couple of commits already on the `master` branch.

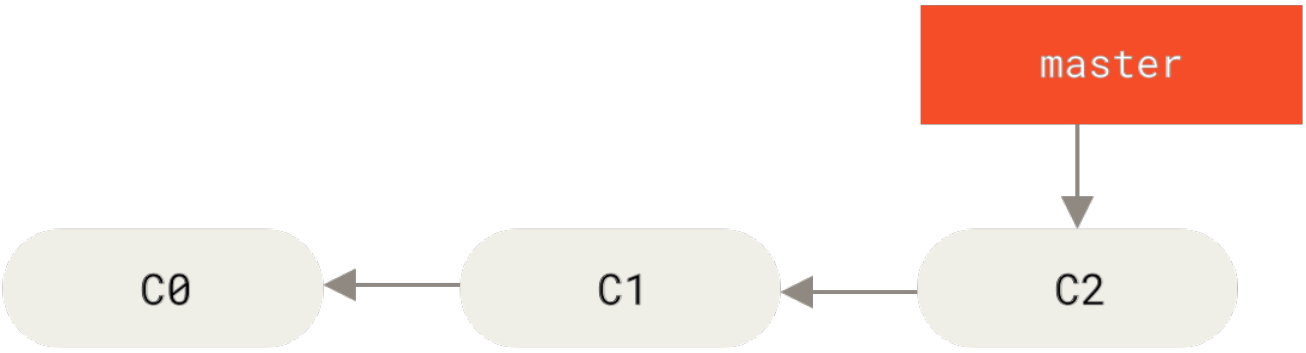

Figure 18. A simple commit history

You've decided that you're going to work on issue #53 in whatever issue-tracking system your company uses. To create a new branch and switch to it at the same time, you can run the `git checkout` command with the `-b` switch:

```
$ git checkout -b iss53
Switched to a new branch "iss53"
```

This is shorthand for:

```
$ git branch iss53
$ git checkout iss53
```

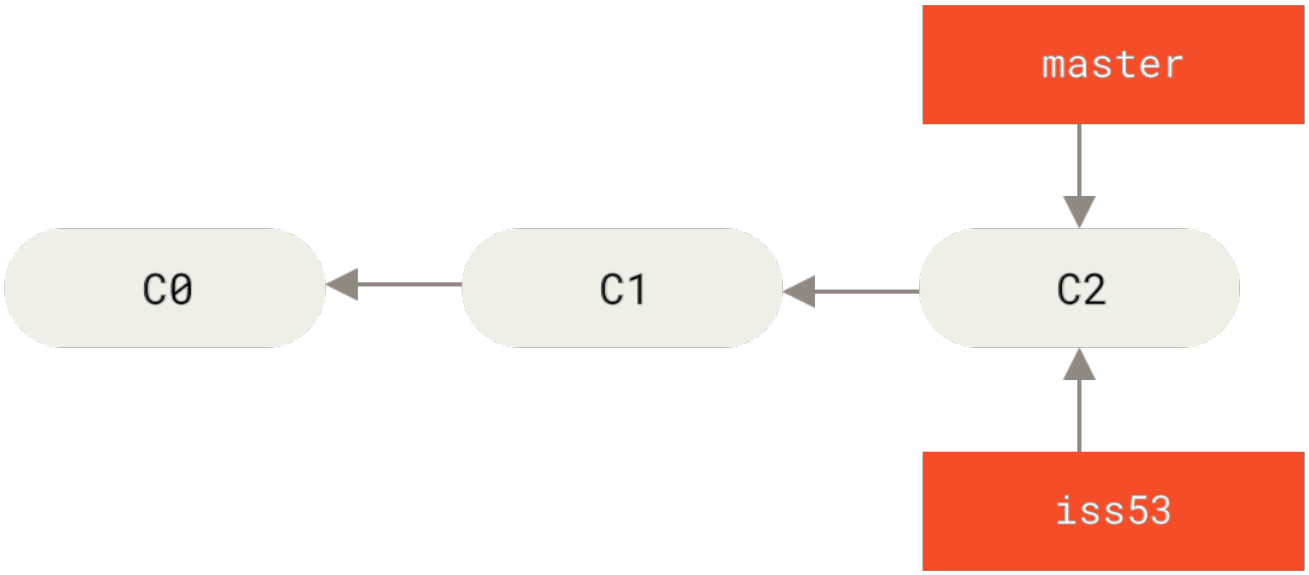

Figure 19. Creating a new branch pointer

You work on your website and do some commits. Doing so moves the `iss53` branch forward, because you have it checked out (that is, your `HEAD` is pointing to it):

```
$ vim index.html
$ git commit -a -m 'Create new footer [issue 53]'
```

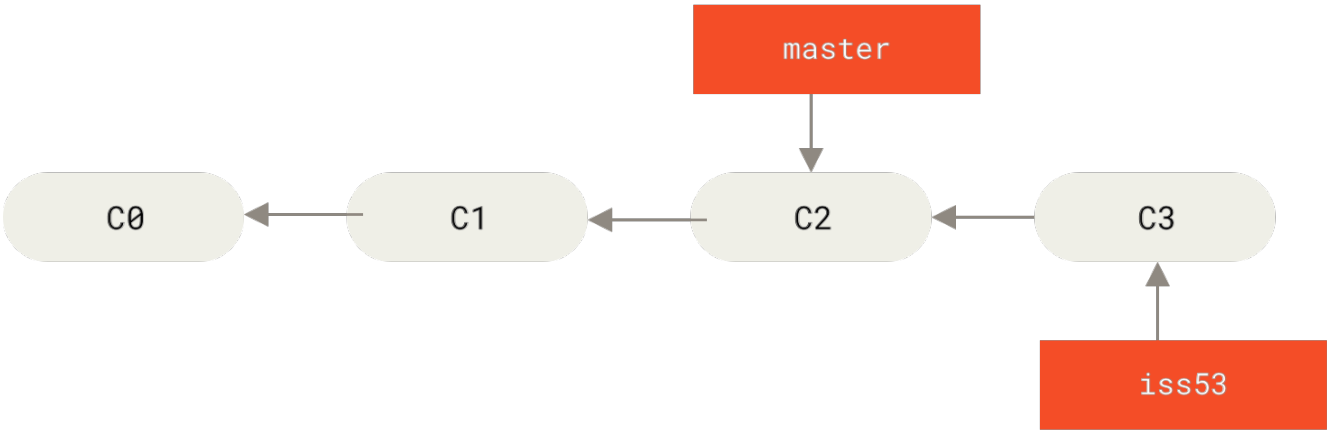

Figure 20. The `iss53` branch has moved forward with your work

Now you get the call that there is an issue with the website, and you need to fix it immediately. With Git, you don't have to deploy your fix along with the `iss53` changes you've made, and you don't have to put a lot of effort into reverting those changes before you can work on applying your fix to what is in production. All you have to do is switch back to your `master` branch.

However, before you do that, note that if your working directory or staging area has uncommitted changes that conflict with the branch you're checking out, Git won't let you switch branches. It's best to have a clean working state when you switch branches. There are ways to get around this (namely, stashing and commit amending) that we'll cover later on, in Stashing and Cleaning. For now, let's assume you've committed all your changes, so you can switch back to your `master` branch:

```
$ git checkout master
Switched to branch 'master'
```

At this point, your project working directory is exactly the way it was before you started working on issue #53, and you can concentrate on your hotfix. This is an important point to remember: when you switch branches, Git resets your working directory to look like it did the last time you committed on that branch. It adds, removes, and modifies files automatically to make sure your working copy is what the branch looked like on your last commit to it.

Next, you have a hotfix to make. Let's create a `hotfix` branch on which to work until it's completed:

```
$ git checkout -b hotfix
Switched to a new branch 'hotfix'
$ vim index.html
$ git commit -a -m 'Fix broken email address'
[hotfix 1fb7853] Fix broken email address
 1 file changed, 2 insertions(+)
```

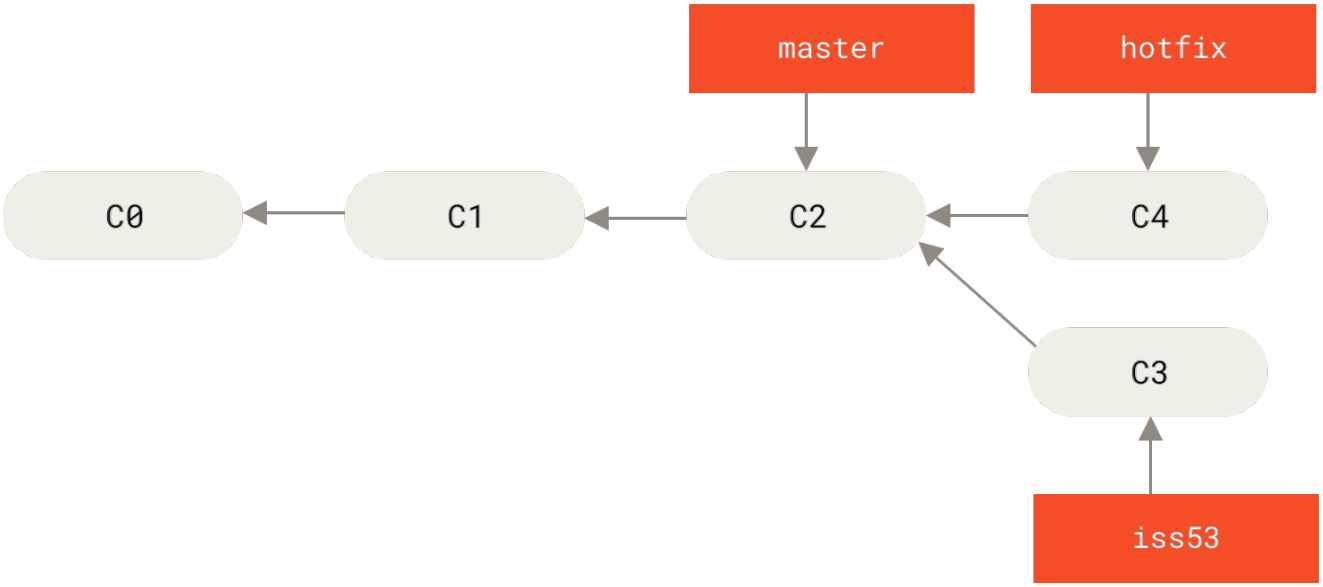

Figure 21. Hotfix branch based on `master`

You can run your tests, make sure the hotfix is what you want, and finally merge the `hotfix` branch back into your `master` branch to deploy to production. You do this with the `git merge` command:

```
$ git checkout master
$ git merge hotfix
Updating f42c576..3a0874c
Fast-forward
 index.html | 2 ++
 1 file changed, 2 insertions(+)
```

You'll notice the phrase "fast-forward" in that merge. Because the commit `C4` pointed to by the branch `hotfix` you merged in was directly ahead of the commit `C2` you're on, Git simply moves the pointer forward. To phrase that another way, when you try to merge one commit with a commit that can be reached by following the first commit's history, Git simplifies things by moving the pointer forward because there is no divergent work to merge together — this is called a "fast-forward."

Your change is now in the snapshot of the commit pointed to by the `master` branch, and you can deploy the fix.

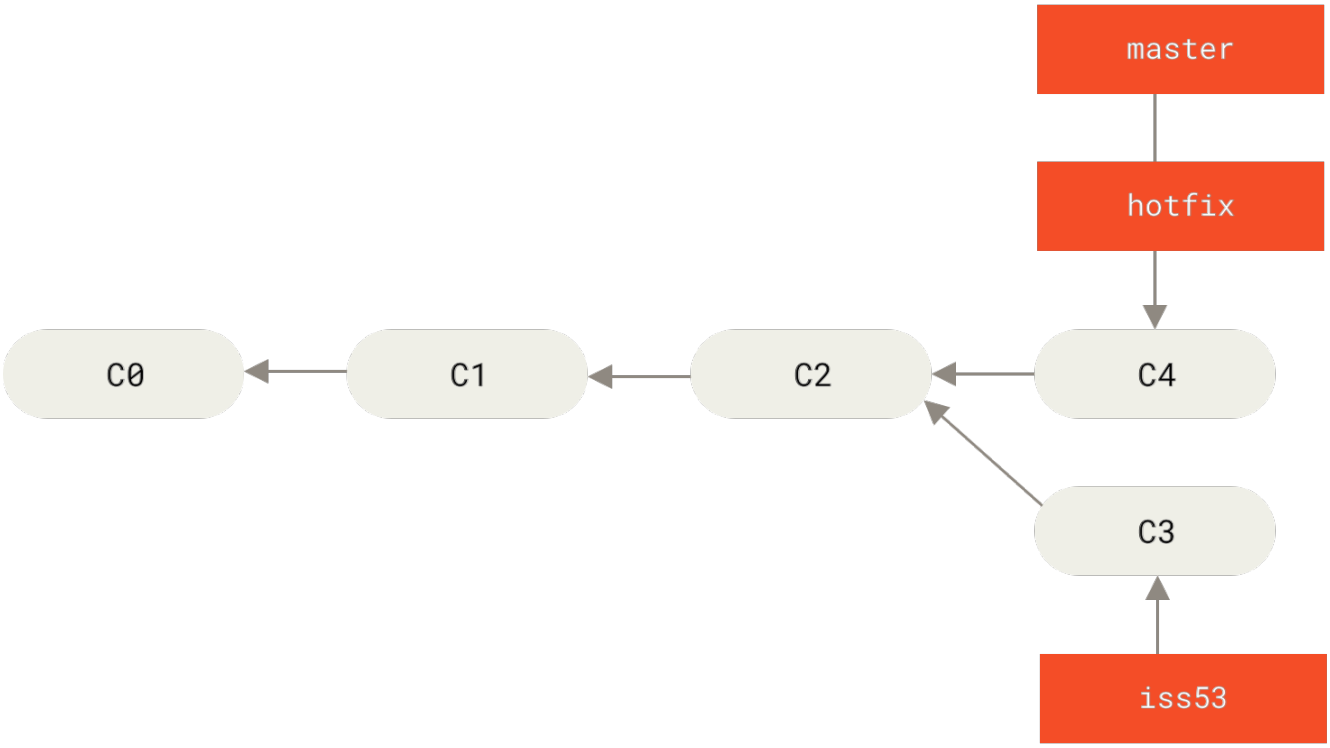

Figure 22. `master` is fast-forwarded to `hotfix`

After your super-important fix is deployed, you're ready to switch back to the work you were doing before you were interrupted. However, first you'll delete the `hotfix` branch, because you no longer need it — the `master` branch points at the same place. You can delete it with the `-d` option to `git branch`:

```
$ git branch -d hotfix
Deleted branch hotfix (3a0874c).
```

Now you can switch back to your work-in-progress branch on issue #53 and continue working on it.

```
$ git checkout iss53
Switched to branch "iss53"
$ vim index.html
$ git commit -a -m 'Finish the new footer [issue 53]'
[iss53 ad82d7a] Finish the new footer [issue 53]
1 file changed, 1 insertion(+)
```

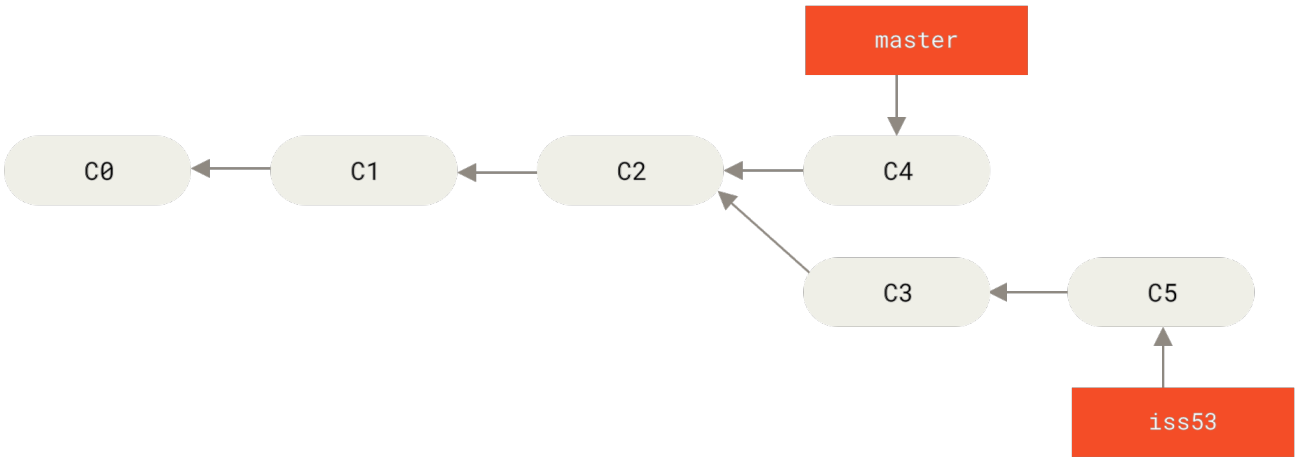

Figure 23. Work continues on `iss53`

It's worth noting here that the work you did in your `hotfix` branch is not contained in the files in your `iss53` branch. If you need to pull it in, you can merge your `master` branch into your `iss53` branch by running `git merge master`, or you can wait to integrate those changes until you decide to pull the `iss53` branch back into `master` later.

Basic Merging

Suppose you've decided that your issue #53 work is complete and ready to be merged into your `master` branch. In order to do that, you'll merge your `iss53` branch into `master`, much like you merged your `hotfix` branch earlier. All you have to do is check out the branch you wish to merge into and then run the `git merge` command:

```
$ git checkout master
Switched to branch 'master'
$ git merge iss53
Merge made by the 'recursive' strategy.
index.html |    1 +
1 file changed, 1 insertion(+)
```

This looks a bit different than the `hotfix` merge you did earlier. In this case, your development history has diverged from some older point. Because the commit on the branch you're on isn't a direct ancestor of the branch you're merging in, Git has to do some work. In this case, Git does a simple three-way merge, using the two snapshots pointed to by the branch tips and the common ancestor of the two.

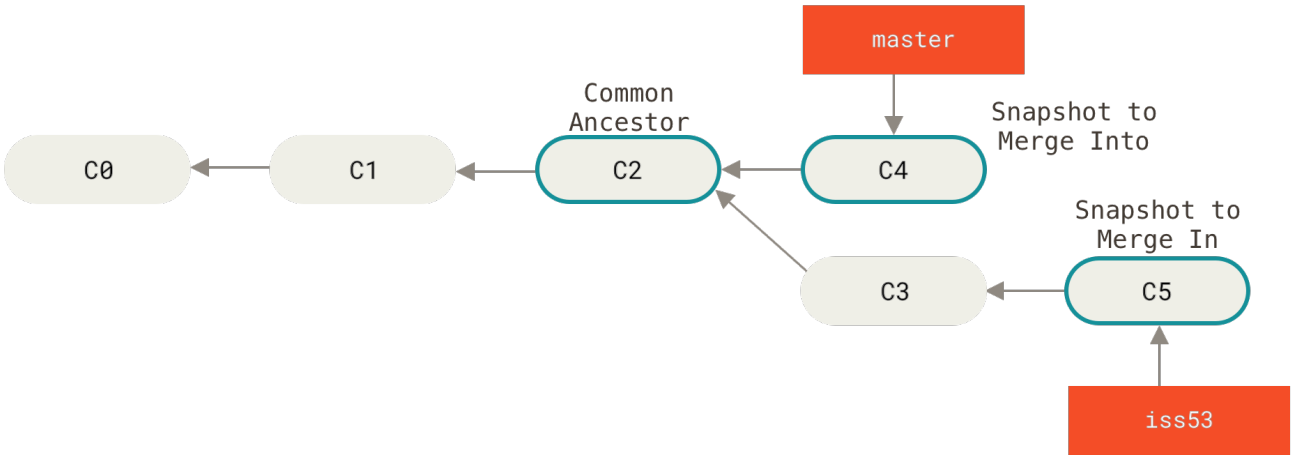

Figure 24. Three snapshots used in a typical merge

Instead of just moving the branch pointer forward, Git creates a new snapshot that results from this three-way merge and automatically creates a new commit that points to it. This is referred to as a merge commit, and is special in that it has more than one parent.

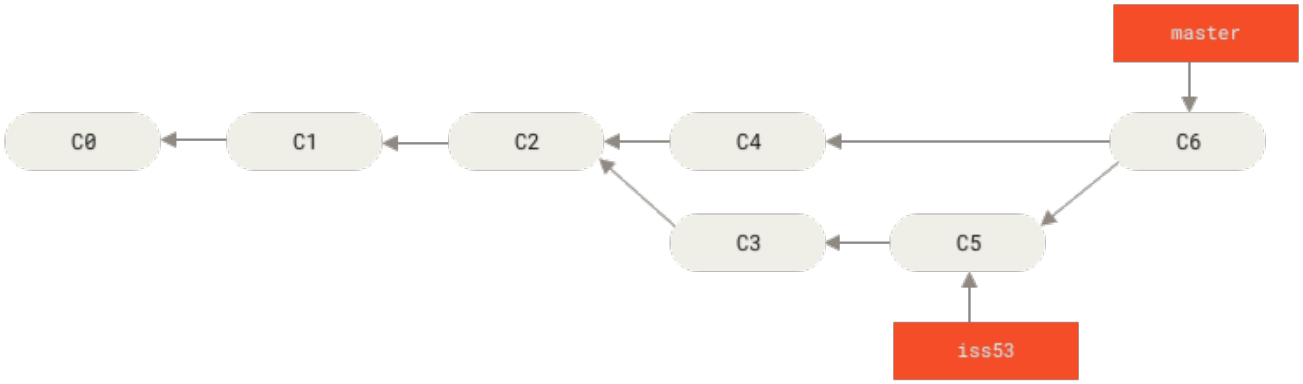

Figure 25. A merge commit

Now that your work is merged in, you have no further need for the `iss53` branch. You can close the issue in your issue-tracking system, and delete the branch:

```
$ git branch -d iss53
```

Basic Merge Conflicts

Occasionally, this process doesn't go smoothly. If you changed the same part of the same file differently in the two branches you're merging, Git won't be able to merge them cleanly. If your fix for issue #53 modified the same part of a file as the `hotfix` branch, you'll get a merge conflict that looks something like this:

```
$ git merge iss53
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

Git hasn't automatically created a new merge commit. It has paused the process while you resolve the conflict. If you want to see which files are 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:      index.html

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

Anything that has merge conflicts and hasn't been resolved is listed as unmerged. Git adds standard conflict-resolution markers to the files that have conflicts, so you can open them manually and resolve those conflicts. Your file contains a section that looks something like this:

```
<<<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=======
<div id="footer">
 please contact us at support@github.com
</div>
>>>>>>> iss53:index.html
```

This means the version in `HEAD` (your `master` branch, because that was what you had checked out when you ran your merge command) is the top part of that block (everything above the `=======`), while the version in your `iss53` branch looks like everything in the bottom part. In order to resolve the conflict, you have to either choose one side or the other or merge the contents yourself. For instance, you might resolve this conflict by replacing the entire block with this:

```
<div id="footer">
please contact us at email.support@github.com
</div>
```

This resolution has a little of each section, and the `<<<<<<<`, `=======`, and `>>>>>>>` lines have been completely removed. After you've resolved each of these sections in each conflicted file, run `git add`

on each file to mark it as resolved. Staging the file marks it as resolved in Git.

If you want to use a graphical tool to resolve these issues, you can run `git mergetool`, which fires up an appropriate visual merge tool and walks you through the conflicts:

```
$ 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:
opendiff kdiff3 tkdiff xxdiff meld tortoisemerge gvimdiff diffuse diffmerge ecmerge
p4merge araxis bc3 codecompare vimdiff emerge
Merging:
index.html

Normal merge conflict for 'index.html':
  {local}: modified file
  {remote}: modified file
Hit return to start merge resolution tool (opendiff):
```

If you want to use a merge tool other than the default (Git chose `opendiff` in this case because the command was run on macOS), you can see all the supported tools listed at the top after “one of the following tools.” Just type the name of the tool you’d rather use.

If you need more advanced tools for resolving tricky merge conflicts, we cover more on merging in Advanced Merging.

After you exit the merge tool, Git asks you if the merge was successful. If you tell the script that it was, it stages the file to mark it as resolved for you. You can run `git status` again to verify that all conflicts have been resolved:

```
$ git status
On branch master
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

Changes to be committed:

    modified:   index.html
```

If you’re happy with that, and you verify that everything that had conflicts has been staged, you can type `git commit` to finalize the merge commit. The commit message by default looks something like this:

```
Merge branch 'iss53'

Conflicts:
```

```
    index.html
#
# 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.
#
# Changes to be committed:
#   modified:   index.html
#
```

If you think it would be helpful to others looking at this merge in the future, you can modify this commit message with details about how you resolved the merge and explain why you did the changes you made if these are not obvious.

Branch Management

Now that you've created, merged, and deleted some branches, let's look at some branch-management tools that will come in handy when you begin using branches all the time.

The `git branch` command does more than just create and delete branches. If you run it with no arguments, you get a simple listing of your current branches:

```
$ git branch
  iss53
* master
  testing
```

Notice the `*` character that prefixes the `master` branch: it indicates the branch that you currently have checked out (i.e., the branch that `HEAD` points to). This means that if you commit at this point, the `master` branch will be moved forward with your new work. To see the last commit on each branch, you can run `git branch -v`:

```
$ git branch -v
  iss53   93b412c Fix javascript issue
* master  7a98805 Merge branch 'iss53'
  testing 782fd34 Add scott to the author list in the readme
```

The useful `--merged` and `--no-merged` options can filter this list to branches that you have or have not yet merged into the branch you're currently on. To see which branches are already merged into the branch you're on, you can run `git branch --merged`:

```
$ git branch --merged
  iss53
* master
```

Because you already merged in `iss53` earlier, you see it in your list. Branches on this list without the `*` in front of them are generally fine to delete with `git branch -d`; you've already incorporated their work into another branch, so you're not going to lose anything.

To see all the branches that contain work you haven't yet merged in, you can run `git branch --no-merged`:

```
$ git branch --no-merged
  testing
```

This shows your other branch. Because it contains work that isn't merged in yet, trying to delete it with `git branch -d` will fail:

```
$ git branch -d testing
error: The branch 'testing' is not fully merged.
If you are sure you want to delete it, run 'git branch -D testing'.
```

If you really do want to delete the branch and lose that work, you can force it with `-D`, as the helpful message points out.

The options described above, `--merged` and `--no-merged` will, if not given a commit or branch name as an argument, show you what is, respectively, merged or not merged into your *current* branch.

You can always provide an additional argument to ask about the merge state with respect to some other branch without checking that other branch out first, as in, what is not merged into the `master` branch?

```
$ git checkout testing
$ git branch --no-merged master
  topicA
  featureB
```

Changing a branch name

Do not rename branches that are still in use by other collaborators. Do not rename a branch like master/main/mainline without having read the section Changing the master branch name.

Suppose you have a branch that is called `bad-branch-name` and you want to change it to `corrected-`

`branch-name`, while keeping all history. You also want to change the branch name on the remote (GitHub, GitLab, other server). How do you do this?

Rename the branch locally with the `git branch --move` command:

```
$ git branch --move bad-branch-name corrected-branch-name
```

This replaces your `bad-branch-name` with `corrected-branch-name`, but this change is only local for now. To let others see the corrected branch on the remote, push it:

```
$ git push --set-upstream origin corrected-branch-name
```

Now we'll take a brief look at where we are now:

```
$ git branch --all
* corrected-branch-name
  main
  remotes/origin/bad-branch-name
  remotes/origin/corrected-branch-name
  remotes/origin/main
```

Notice that you're on the branch `corrected-branch-name` and it's available on the remote. However, the branch with the bad name is also still present there but you can delete it by executing the following command:

```
$ git push origin --delete bad-branch-name
```

Now the bad branch name is fully replaced with the corrected branch name.

Changing the master branch name

Changing the name of a branch like master/main/mainline/default will break the integrations, services, helper utilities and build/release scripts that your repository uses. Before you do this, make sure you consult with your collaborators. Also, make sure you do a thorough search through your repo and update any references to the old branch name in your code and scripts.

Rename your local `master` branch into `main` with the following command:

```
$ git branch --move master main
```

There's no local `master` branch anymore, because it's renamed to the `main` branch.

To let others see the new `main` branch, you need to push it to the remote. This makes the renamed

branch available on the remote.

```
$ git push --set-upstream origin main
```

Now we end up with the following state:

```
$ git branch --all
* main
  remotes/origin/HEAD -> origin/master
  remotes/origin/main
  remotes/origin/master
```

Your local `master` branch is gone, as it's replaced with the `main` branch. The `main` branch is present on the remote. However, the old `master` branch is still present on the remote. Other collaborators will continue to use the `master` branch as the base of their work, until you make some further changes.

Now you have a few more tasks in front of you to complete the transition:

- Any projects that depend on this one will need to update their code and/or configuration.
- Update any test-runner configuration files.
- Adjust build and release scripts.
- Redirect settings on your repo host for things like the repo's default branch, merge rules, and other things that match branch names.
- Update references to the old branch in documentation.
- Close or merge any pull requests that target the old branch.

After you've done all these tasks, and are certain the `main` branch performs just as the `master` branch, you can delete the `master` branch:

```
$ git push origin --delete master
```

Branching Workflows

Now that you have the basics of branching and merging down, what can or should you do with them? In this section, we'll cover some common workflows that this lightweight branching makes possible, so you can decide if you would like to incorporate them into your own development cycle.

Long-Running Branches

Because Git uses a simple three-way merge, merging from one branch into another multiple times over a long period is generally easy to do. This means you can have several branches that are always open and that you use for different stages of your development cycle; you can merge regularly from some of them into others.

Many Git developers have a workflow that embraces this approach, such as having only code that is entirely stable in their `master` branch — possibly only code that has been or will be released. They have another parallel branch named `develop` or `next` that they work from or use to test stability — it isn't necessarily always stable, but whenever it gets to a stable state, it can be merged into `master`. It's used to pull in topic branches (short-lived branches, like your earlier `iss53` branch) when they're ready, to make sure they pass all the tests and don't introduce bugs.

In reality, we're talking about pointers moving up the line of commits you're making. The stable branches are farther down the line in your commit history, and the bleeding-edge branches are farther up the history.

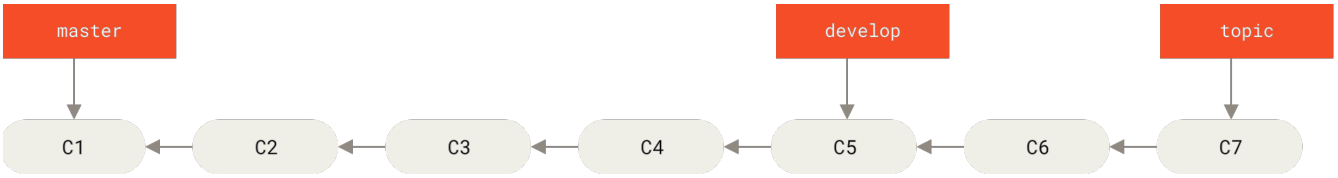

Figure 26. A linear view of progressive-stability branching

It's generally easier to think about them as work silos, where sets of commits graduate to a more stable silo when they're fully tested.

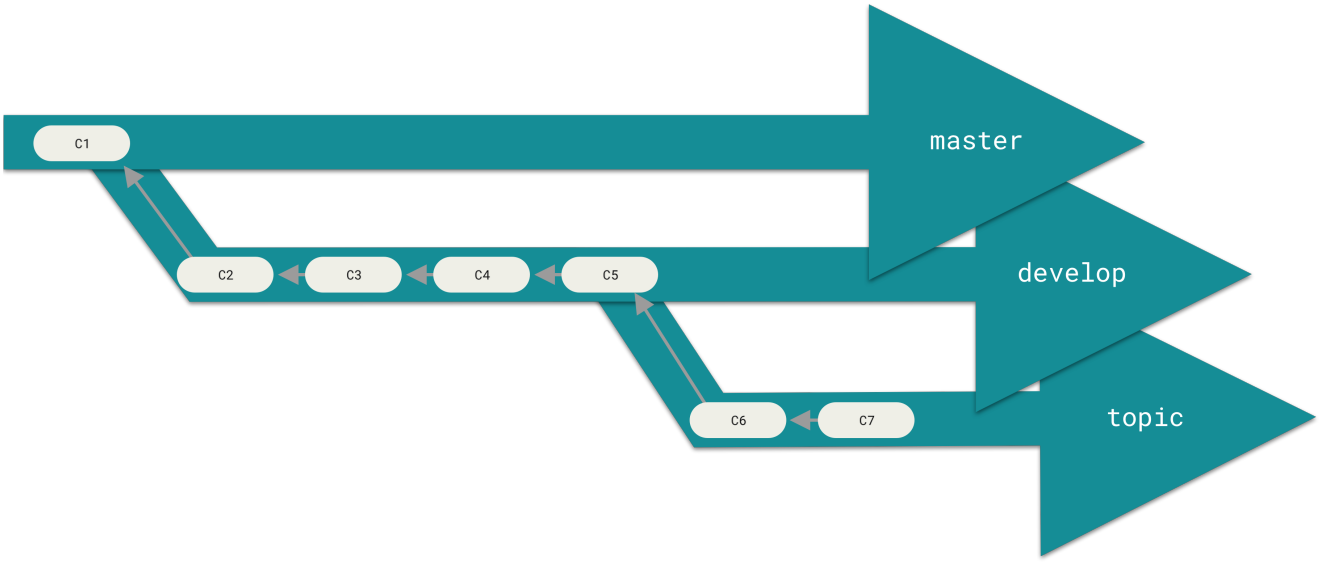

Figure 27. A "silo" view of progressive-stability branching

You can keep doing this for several levels of stability. Some larger projects also have a `proposed` or `pu` (proposed updates) branch that has integrated branches that may not be ready to go into the `next` or `master` branch. The idea is that your branches are at various levels of stability; when they reach a more stable level, they're merged into the branch above them. Again, having multiple long-running branches isn't necessary, but it's often helpful, especially when you're dealing with very large or complex projects.

Topic Branches

Topic branches, however, are useful in projects of any size. A topic branch is a short-lived branch that you create and use for a single particular feature or related work. This is something you've likely never done with a VCS before because it's generally too expensive to create and merge

branches. But in Git it's common to create, work on, merge, and delete branches several times a day.

You saw this in the last section with the `iss53` and `hotfix` branches you created. You did a few commits on them and deleted them directly after merging them into your main branch. This technique allows you to context-switch quickly and completely — because your work is separated into silos where all the changes in that branch have to do with that topic, it's easier to see what has happened during code review and such. You can keep the changes there for minutes, days, or months, and merge them in when they're ready, regardless of the order in which they were created or worked on.

Consider an example of doing some work (on `master`), branching off for an issue (`iss91`), working on it for a bit, branching off the second branch to try another way of handling the same thing (`iss91v2`), going back to your `master` branch and working there for a while, and then branching off there to do some work that you're not sure is a good idea (`dumbidea` branch). Your commit history will look something like this:

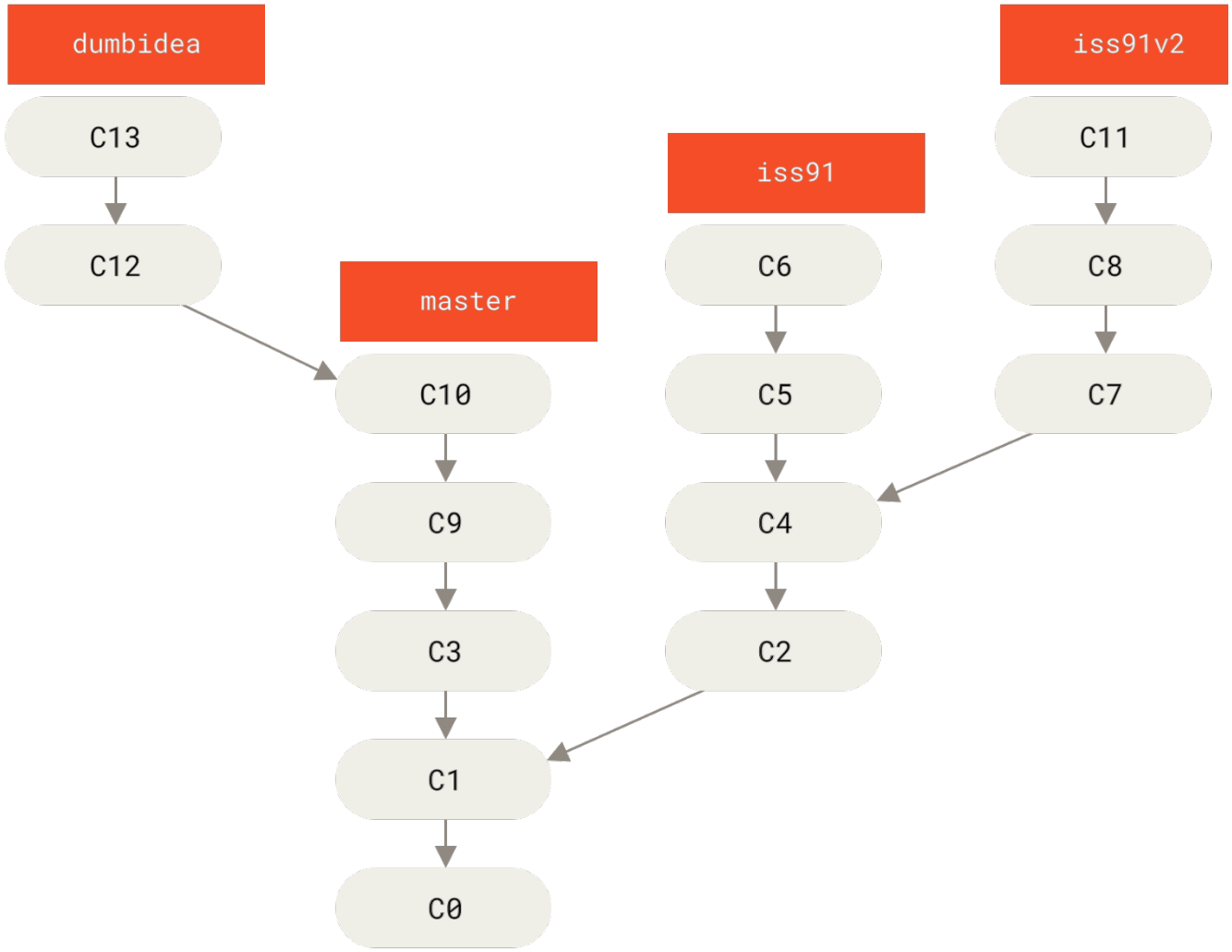

Figure 28. Multiple topic branches

Now, let's say you decide you like the second solution to your issue best (`iss91v2`); and you showed the `dumbidea` branch to your coworkers, and it turns out to be genius. You can throw away the original `iss91` branch (losing commits `C5` and `C6`) and merge in the other two. Your history then looks like this:

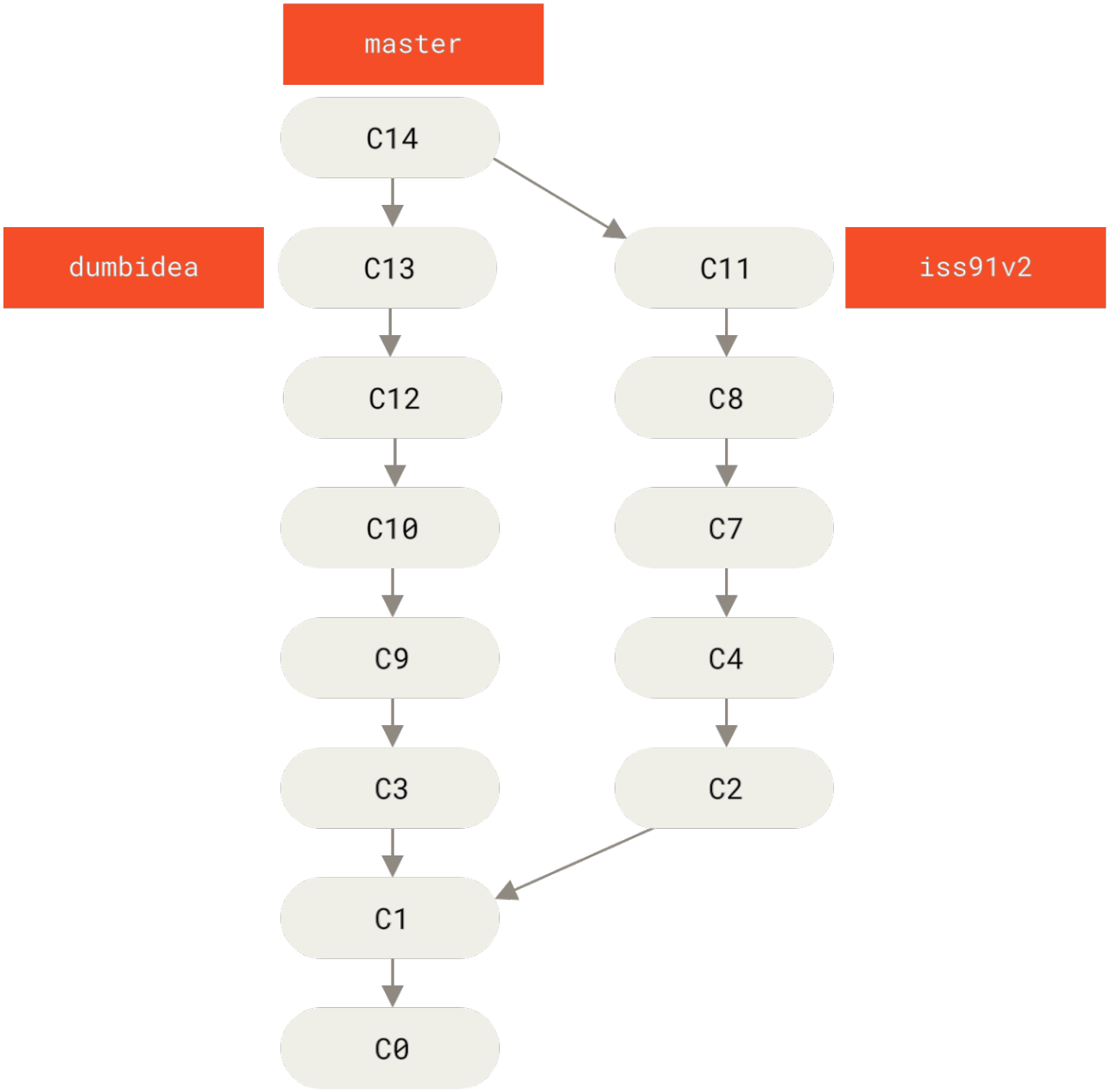

Figure 29. History after merging `dumbidea` *and* `iss91v2`

We will go into more detail about the various possible workflows for your Git project in Distributed Git, so before you decide which branching scheme your next project will use, be sure to read that chapter.

It's important to remember when you're doing all this that these branches are completely local. When you're branching and merging, everything is being done only in your Git repository — there is no communication with the server.

Remote Branches

Remote references are references (pointers) in your remote repositories, including branches, tags, and so on. You can get a full list of remote references explicitly with `git ls-remote <remote>`, or `git remote show <remote>` for remote branches as well as more information. Nevertheless, a more common way is to take advantage of remote-tracking branches.

Remote-tracking branches are references to the state of remote branches. They're local references that you can't move; Git moves them for you whenever you do any network communication, to make sure they accurately represent the state of the remote repository. Think of them as bookmarks, to remind you where the branches in your remote repositories were the last time you connected to them.

Remote-tracking branch names take the form `<remote>/<branch>`. For instance, if you wanted to see what the `master` branch on your `origin` remote looked like as of the last time you communicated with it, you would check the `origin/master` branch. If you were working on an issue with a partner and they pushed up an `iss53` branch, you might have your own local `iss53` branch, but the branch on the server would be represented by the remote-tracking branch `origin/iss53`.

This may be a bit confusing, so let's look at an example. Let's say you have a Git server on your network at `git.ourcompany.com`. If you clone from this, Git's `clone` command automatically names it `origin` for you, pulls down all its data, creates a pointer to where its `master` branch is, and names it `origin/master` locally. Git also gives you your own local `master` branch starting at the same place as origin's `master` branch, so you have something to work from.

"origin" is not special

Just like the branch name "master" does not have any special meaning in Git, neither does "origin". While "master" is the default name for a starting branch when you run `git init` which is the only reason it's widely used, "origin" is the default name for a remote when you run `git clone`. If you run `git clone -o booyah` instead, then you will have `booyah/master` as your default remote branch.

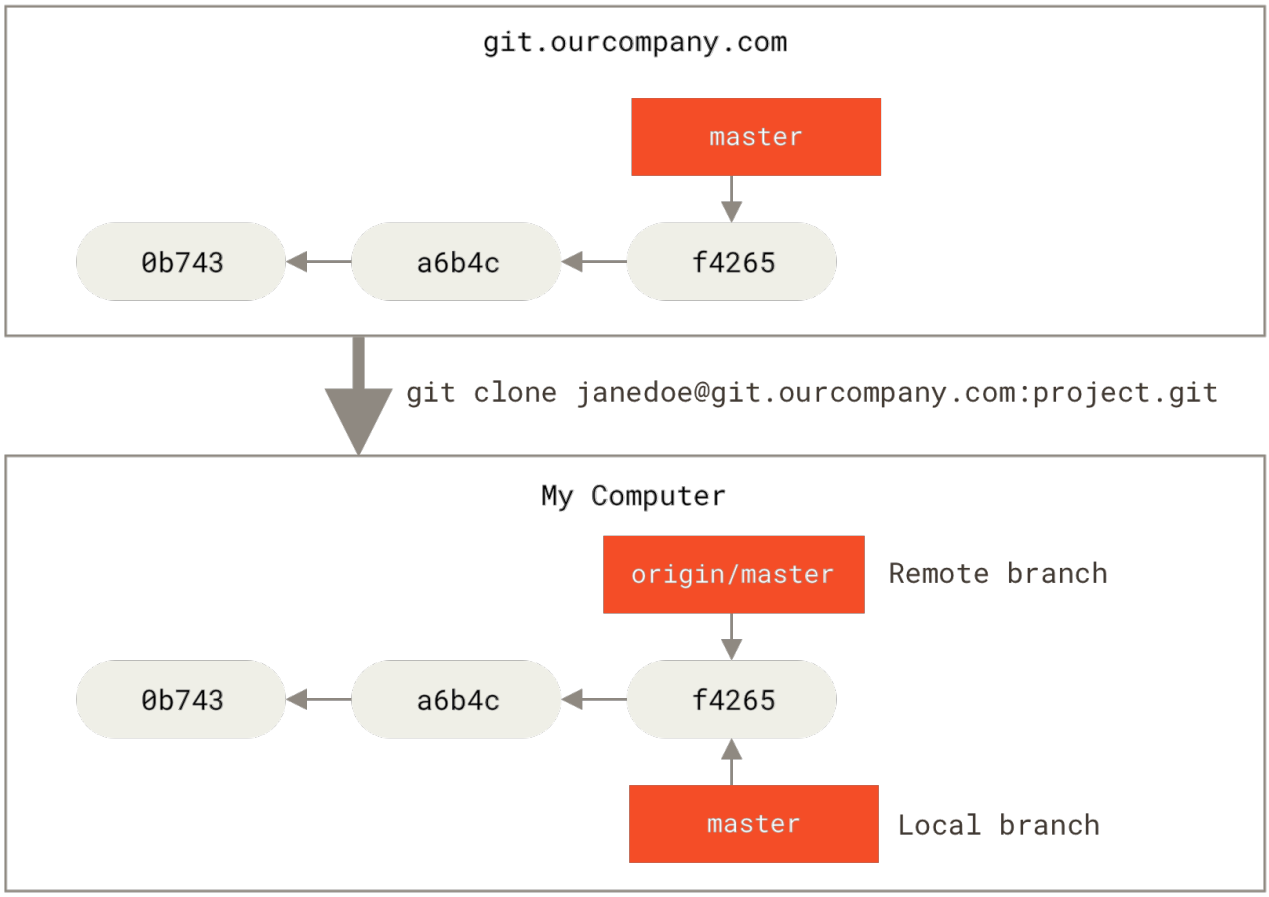

Figure 30. Server and local repositories after cloning

If you do some work on your local `master` branch, and, in the meantime, someone else pushes to `git.ourcompany.com` and updates its `master` branch, then your histories move forward differently. Also, as long as you stay out of contact with your `origin` server, your `origin/master` pointer doesn't move.

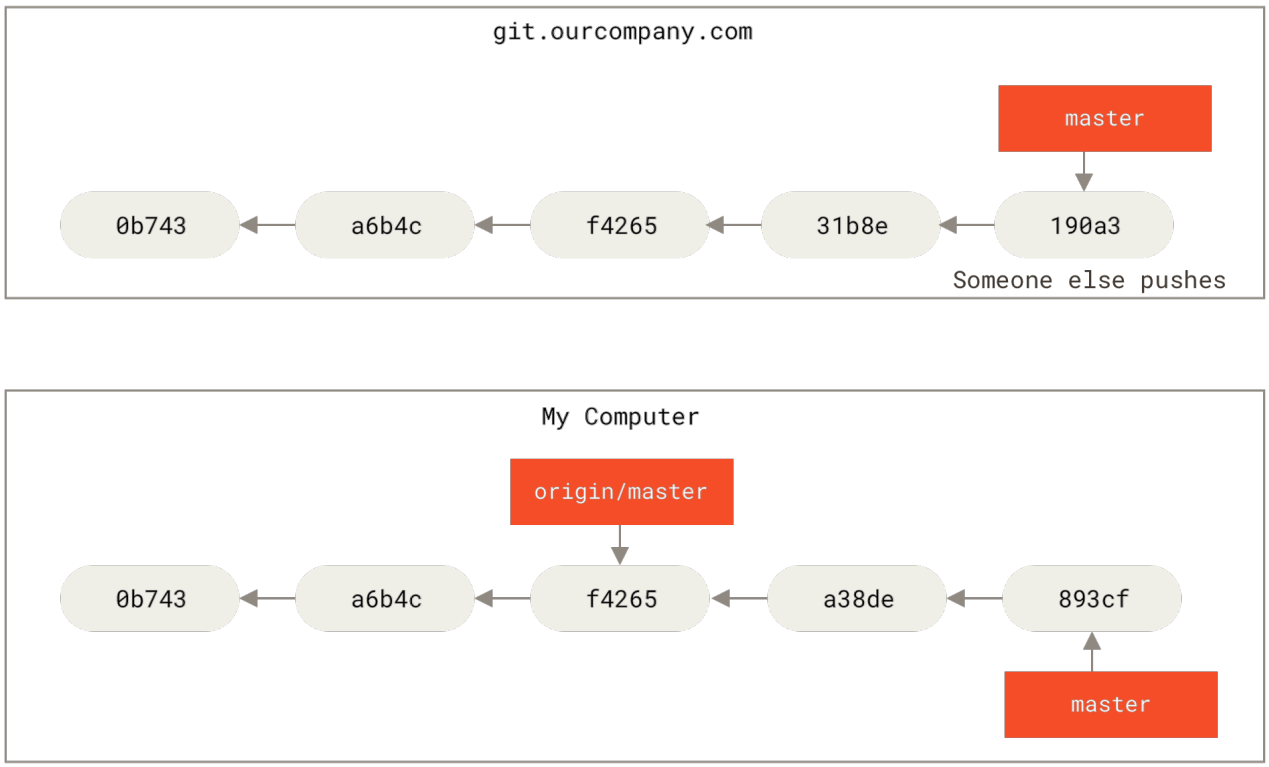

Figure 31. Local and remote work can diverge

To synchronize your work with a given remote, you run a `git fetch <remote>` command (in our case, `git fetch origin`). This command looks up which server "origin" is (in this case, it's `git.ourcompany.com`), fetches any data from it that you don't yet have, and updates your local database, moving your `origin/master` pointer to its new, more up-to-date position.

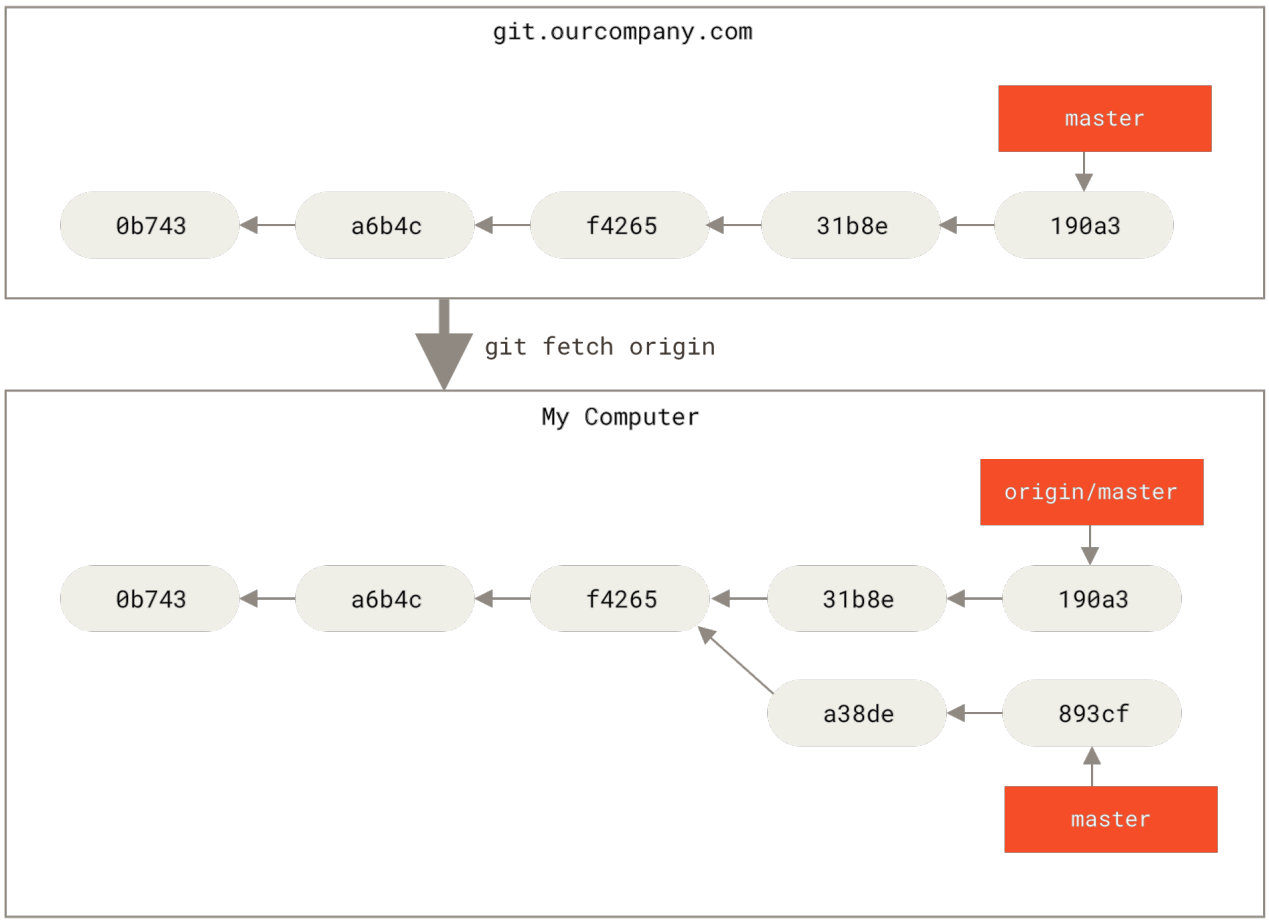

Figure 32. `git fetch` updates your remote-tracking branches

To demonstrate having multiple remote servers and what remote branches for those remote projects look like, let's assume you have another internal Git server that is used only for development by one of your sprint teams. This server is at `git.team1.ourcompany.com`. You can add it as a new remote reference to the project you're currently working on by running the `git remote add` command as we covered in Git Basics. Name this remote `teamone`, which will be your shortname for that whole URL.

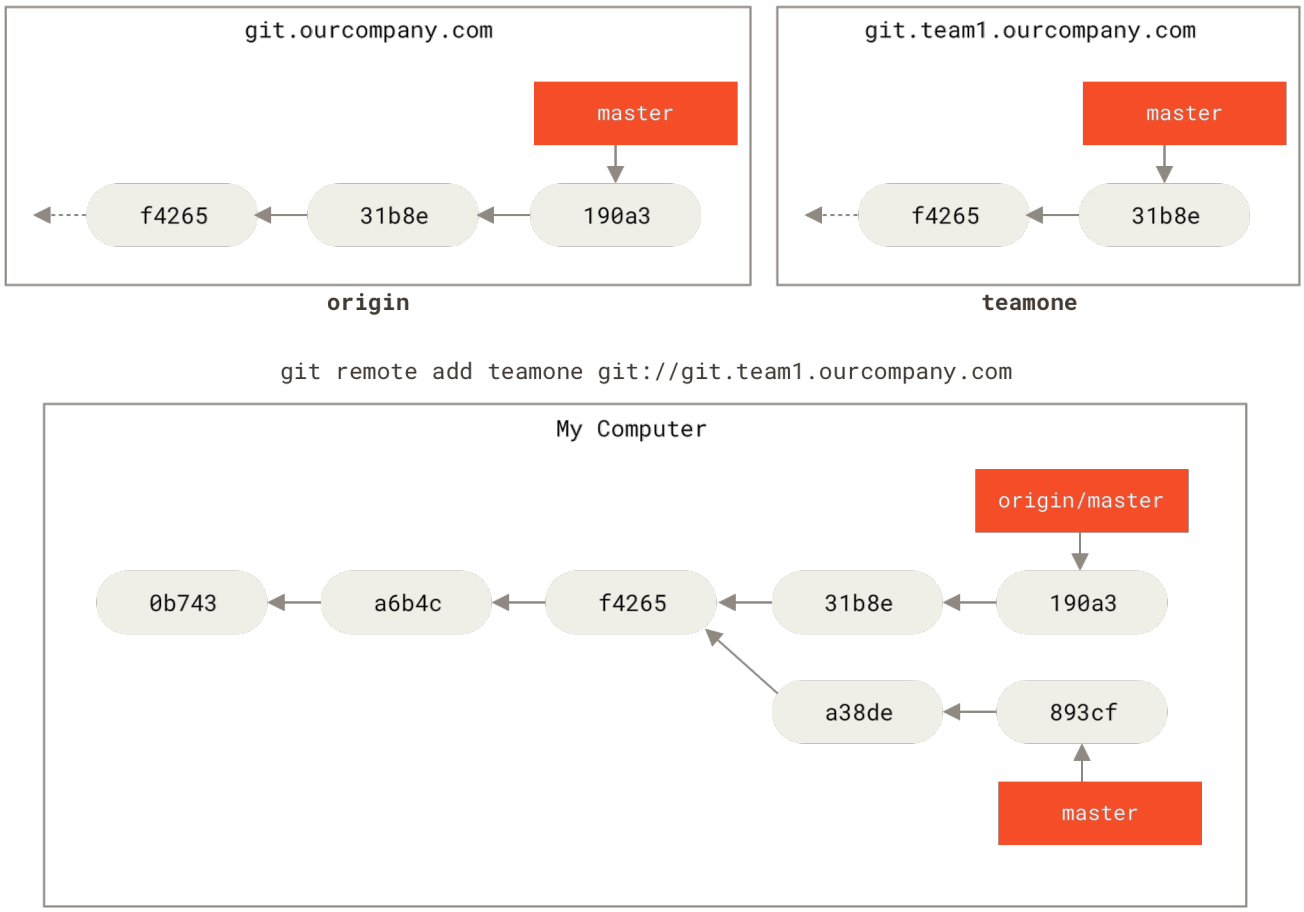

Figure 33. Adding another server as a remote

Now, you can run `git fetch teamone` to fetch everything the remote `teamone` server has that you don't have yet. Because that server has a subset of the data your `origin` server has right now, Git fetches no data but sets a remote-tracking branch called `teamone/master` to point to the commit that `teamone` has as its `master` branch.

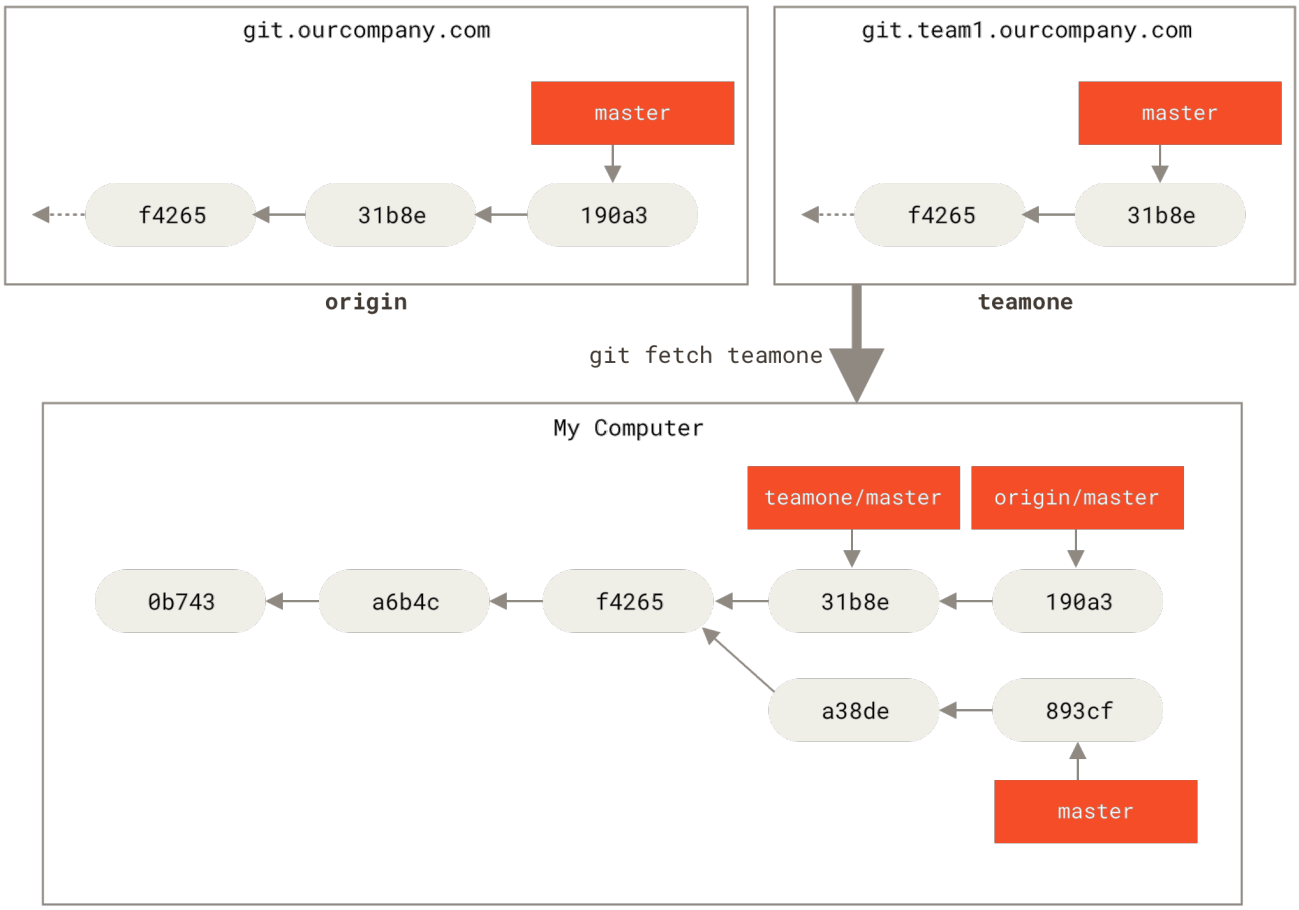

Figure 34. Remote-tracking branch for `teamone/master`

Pushing

When you want to share a branch with the world, you need to push it up to a remote to which you have write access. Your local branches aren't automatically synchronized to the remotes you write to — you have to explicitly push the branches you want to share. That way, you can use private branches for work you don't want to share, and push up only the topic branches you want to collaborate on.

If you have a branch named `serverfix` that you want to work on with others, you can push it up the same way you pushed your first branch. Run `git push <remote> <branch>`:

```
$ git push origin serverfix
Counting objects: 24, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (15/15), done.
Writing objects: 100% (24/24), 1.91 KiB | 0 bytes/s, done.
Total 24 (delta 2), reused 0 (delta 0)
To https://github.com/schacon/simplegit
 * [new branch]      serverfix -> serverfix
```

This is a bit of a shortcut. Git automatically expands the `serverfix` branchname out to `refs/heads/serverfix:refs/heads/serverfix`, which means, "Take my `serverfix` local branch and push it to update the remote's `serverfix` branch." We'll go over the `refs/heads/` part in detail in Git

Internals, but you can generally leave it off. You can also do `git push origin serverfix:serverfix`, which does the same thing — it says, "Take my serverfix and make it the remote's serverfix." You can use this format to push a local branch into a remote branch that is named differently. If you didn't want it to be called `serverfix` on the remote, you could instead run `git push origin serverfix:awesomebranch` to push your local `serverfix` branch to the `awesomebranch` branch on the remote project.

Don't type your password every time

If you're using an HTTPS URL to push over, the Git server will ask you for your username and password for authentication. By default it will prompt you on the terminal for this information so the server can tell if you're allowed to push.

If you don't want to type it every single time you push, you can set up a "credential cache". The simplest is just to keep it in memory for a few minutes, which you can easily set up by running `git config --global credential.helper cache`.

For more information on the various credential caching options available, see Credential Storage.

The next time one of your collaborators fetches from the server, they will get a reference to where the server's version of `serverfix` is under the remote branch `origin/serverfix`:

```
$ git fetch origin
remote: Counting objects: 7, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 3 (delta 0)
Unpacking objects: 100% (3/3), done.
From https://github.com/schacon/simplegit
 * [new branch]      serverfix    -> origin/serverfix
```

It's important to note that when you do a fetch that brings down new remote-tracking branches, you don't automatically have local, editable copies of them. In other words, in this case, you don't have a new `serverfix` branch — you have only an `origin/serverfix` pointer that you can't modify.

To merge this work into your current working branch, you can run `git merge origin/serverfix`. If you want your own `serverfix` branch that you can work on, you can base it off your remote-tracking branch:

```
$ git checkout -b serverfix origin/serverfix
Branch serverfix set up to track remote branch serverfix from origin.
Switched to a new branch 'serverfix'
```

This gives you a local branch that you can work on that starts where `origin/serverfix` is.

Tracking Branches

Checking out a local branch from a remote-tracking branch automatically creates what is called a

“tracking branch” (and the branch it tracks is called an “upstream branch”). Tracking branches are local branches that have a direct relationship to a remote branch. If you’re on a tracking branch and type `git pull`, Git automatically knows which server to fetch from and which branch to merge in.

When you clone a repository, it generally automatically creates a `master` branch that tracks `origin/master`. However, you can set up other tracking branches if you wish — ones that track branches on other remotes, or don’t track the `master` branch. The simple case is the example you just saw, running `git checkout -b <branch> <remote>/<branch>`. This is a common enough operation that Git provides the `--track` shorthand:

```
$ git checkout --track origin/serverfix
Branch serverfix set up to track remote branch serverfix from origin.
Switched to a new branch 'serverfix'
```

In fact, this is so common that there’s even a shortcut for that shortcut. If the branch name you’re trying to checkout (a) doesn’t exist and (b) exactly matches a name on only one remote, Git will create a tracking branch for you:

```
$ git checkout serverfix
Branch serverfix set up to track remote branch serverfix from origin.
Switched to a new branch 'serverfix'
```

To set up a local branch with a different name than the remote branch, you can easily use the first version with a different local branch name:

```
$ git checkout -b sf origin/serverfix
Branch sf set up to track remote branch serverfix from origin.
Switched to a new branch 'sf'
```

Now, your local branch `sf` will automatically pull from `origin/serverfix`.

If you already have a local branch and want to set it to a remote branch you just pulled down, or want to change the upstream branch you’re tracking, you can use the `-u` or `--set-upstream-to` option to `git branch` to explicitly set it at any time.

```
$ git branch -u origin/serverfix
Branch serverfix set up to track remote branch serverfix from origin.
```

Upstream shorthand

When you have a tracking branch set up, you can reference its upstream branch with the `@{upstream}` or `@{u}` shorthand. So if you’re on the `master` branch and it’s tracking `origin/master`, you can say something like `git merge @{u}` instead of `git merge origin/master` if you wish.

If you want to see what tracking branches you have set up, you can use the `-vv` option to `git branch`. This will list out your local branches with more information including what each branch is tracking and if your local branch is ahead, behind or both.

```
$ git branch -vv
  iss53     7e424c3 [origin/iss53: ahead 2] Add forgotten brackets
  master    1ae2a45 [origin/master] Deploy index fix
* serverfix f8674d9 [teamone/server-fix-good: ahead 3, behind 1] This should do it
  testing   5ea463a Try something new
```

So here we can see that our `iss53` branch is tracking `origin/iss53` and is "ahead" by two, meaning that we have two commits locally that are not pushed to the server. We can also see that our `master` branch is tracking `origin/master` and is up to date. Next we can see that our `serverfix` branch is tracking the `server-fix-good` branch on our `teamone` server and is ahead by three and behind by one, meaning that there is one commit on the server we haven't merged in yet and three commits locally that we haven't pushed. Finally we can see that our `testing` branch is not tracking any remote branch.

It's important to note that these numbers are only since the last time you fetched from each server. This command does not reach out to the servers, it's telling you about what it has cached from these servers locally. If you want totally up to date ahead and behind numbers, you'll need to fetch from all your remotes right before running this. You could do that like this:

```
$ git fetch --all; git branch -vv
```

Pulling

While the `git fetch` command will fetch all the changes on the server that you don't have yet, it will not modify your working directory at all. It will simply get the data for you and let you merge it yourself. However, there is a command called `git pull` which is essentially a `git fetch` immediately followed by a `git merge` in most cases. If you have a tracking branch set up as demonstrated in the last section, either by explicitly setting it or by having it created for you by the `clone` or `checkout` commands, `git pull` will look up what server and branch your current branch is tracking, fetch from that server and then try to merge in that remote branch.

Deleting Remote Branches

Suppose you're done with a remote branch — say you and your collaborators are finished with a feature and have merged it into your remote's `master` branch (or whatever branch your stable codeline is in). You can delete a remote branch using the `--delete` option to `git push`. If you want to delete your `serverfix` branch from the server, you run the following:

```
$ git push origin --delete serverfix
To https://github.com/schacon/simplegit
 - [deleted]         serverfix
```

Basically all this does is to remove the pointer from the server. The Git server will generally keep the data there for a while until a garbage collection runs, so if it was accidentally deleted, it's often easy to recover.

Rebasing

In Git, there are two main ways to integrate changes from one branch into another: the `merge` and the `rebase`. In this section you'll learn what rebasing is, how to do it, why it's a pretty amazing tool, and in what cases you won't want to use it.

The Basic Rebase

If you go back to an earlier example from Basic Merging, you can see that you diverged your work and made commits on two different branches.

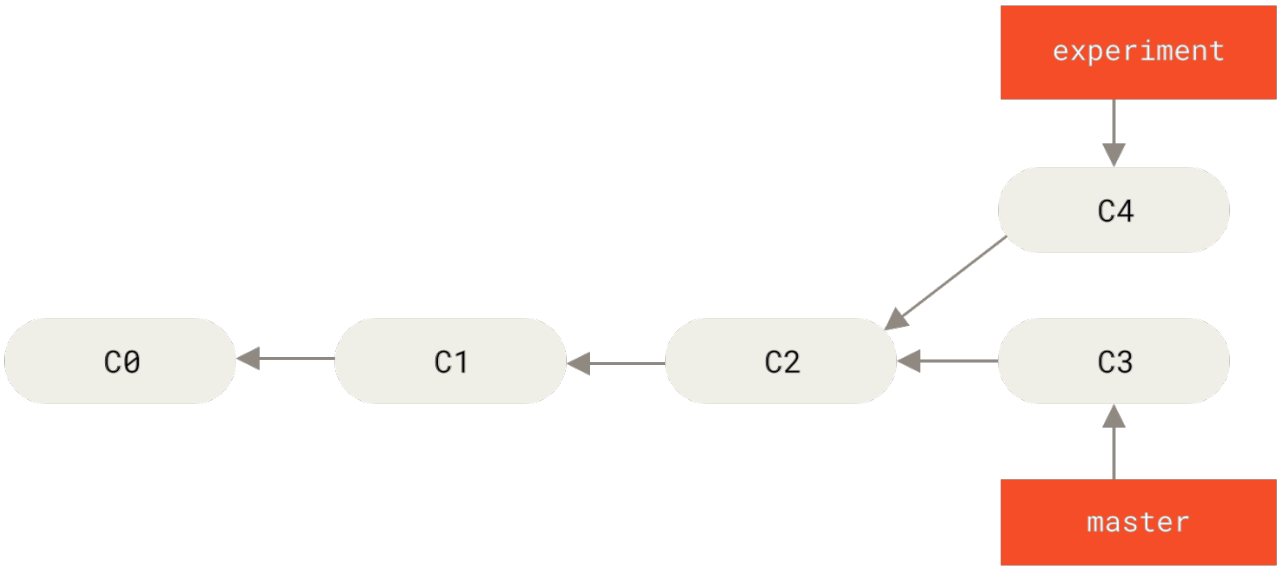

Figure 35. Simple divergent history

The easiest way to integrate the branches, as we've already covered, is the `merge` command. It performs a three-way merge between the two latest branch snapshots (`C3` and `C4`) and the most recent common ancestor of the two (`C2`), creating a new snapshot (and commit).

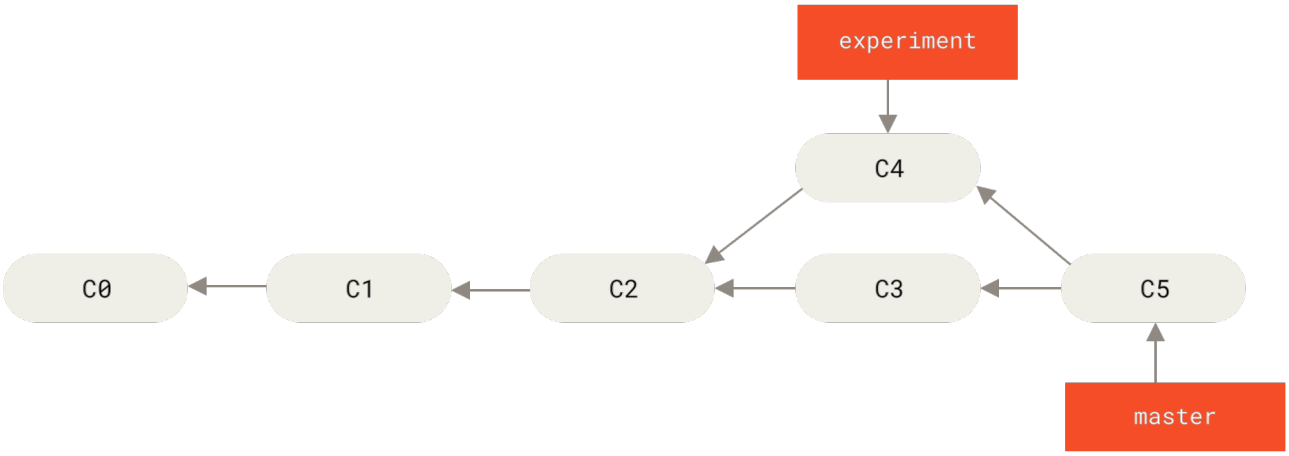

Figure 36. Merging to integrate diverged work history

However, there is another way: you can take the patch of the change that was introduced in `C4` and reapply it on top of `C3`. In Git, this is called *rebasing*. With the `rebase` command, you can take all the changes that were committed on one branch and replay them on a different branch.

For this example, you would check out the `experiment` branch, and then rebase it onto the `master` branch as follows:

```
$ git checkout experiment
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: added staged command
```

This operation works by going to the common ancestor of the two branches (the one you're on and the one you're rebasing onto), getting the diff introduced by each commit of the branch you're on, saving those diffs to temporary files, resetting the current branch to the same commit as the branch you are rebasing onto, and finally applying each change in turn.

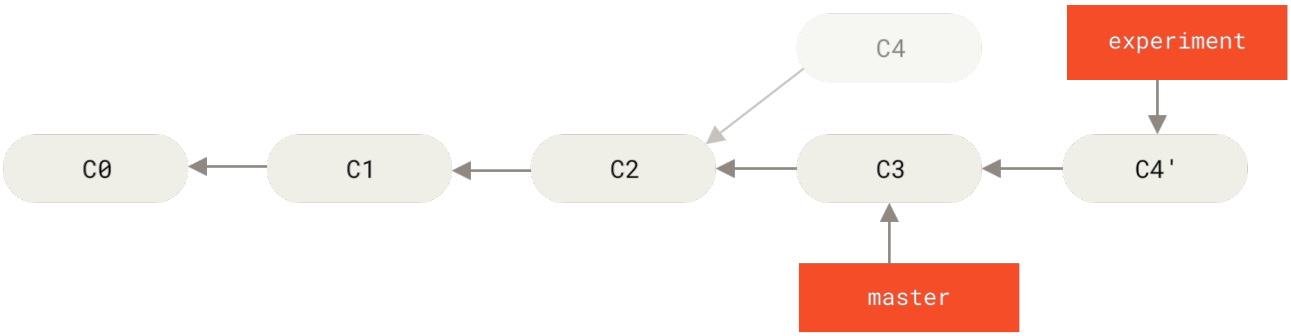

Figure 37. Rebasing the change introduced in `C4` onto `C3`

At this point, you can go back to the `master` branch and do a fast-forward merge.

```
$ git checkout master
$ git merge experiment
```

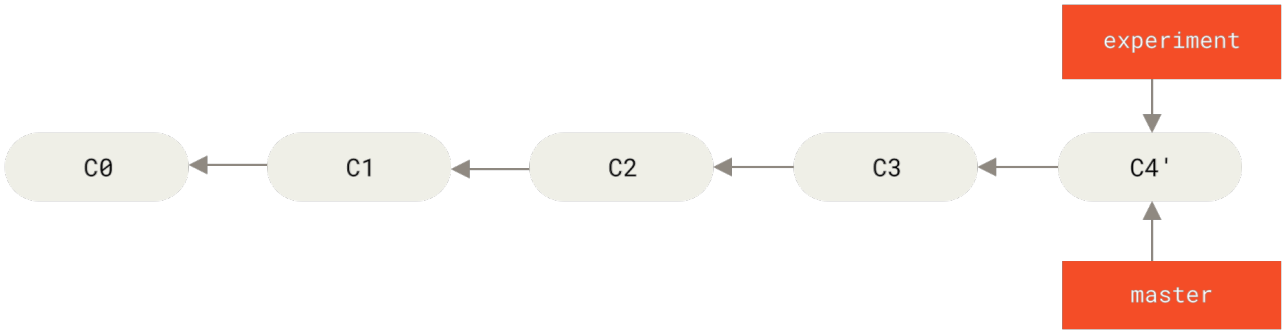

Figure 38. Fast-forwarding the `master` branch

Now, the snapshot pointed to by `C4'` is exactly the same as the one that was pointed to by `C5` in the merge example. There is no difference in the end product of the integration, but rebasing makes for a cleaner history. If you examine the log of a rebased branch, it looks like a linear history: it appears that all the work happened in series, even when it originally happened in parallel.

Often, you'll do this to make sure your commits apply cleanly on a remote branch — perhaps in a project to which you're trying to contribute but that you don't maintain. In this case, you'd do your work in a branch and then rebase your work onto `origin/master` when you were ready to submit your patches to the main project. That way, the maintainer doesn't have to do any integration work — just a fast-forward or a clean apply.

Note that the snapshot pointed to by the final commit you end up with, whether it's the last of the rebased commits for a rebase or the final merge commit after a merge, is the same snapshot — it's only the history that is different. Rebasing replays changes from one line of work onto another in the order they were introduced, whereas merging takes the endpoints and merges them together.

More Interesting Rebases

You can also have your rebase replay on something other than the rebase target branch. Take a history like A history with a topic branch off another topic branch, for example. You branched a topic branch (`server`) to add some server-side functionality to your project, and made a commit. Then, you branched off that to make the client-side changes (`client`) and committed a few times. Finally, you went back to your `server` branch and did a few more commits.

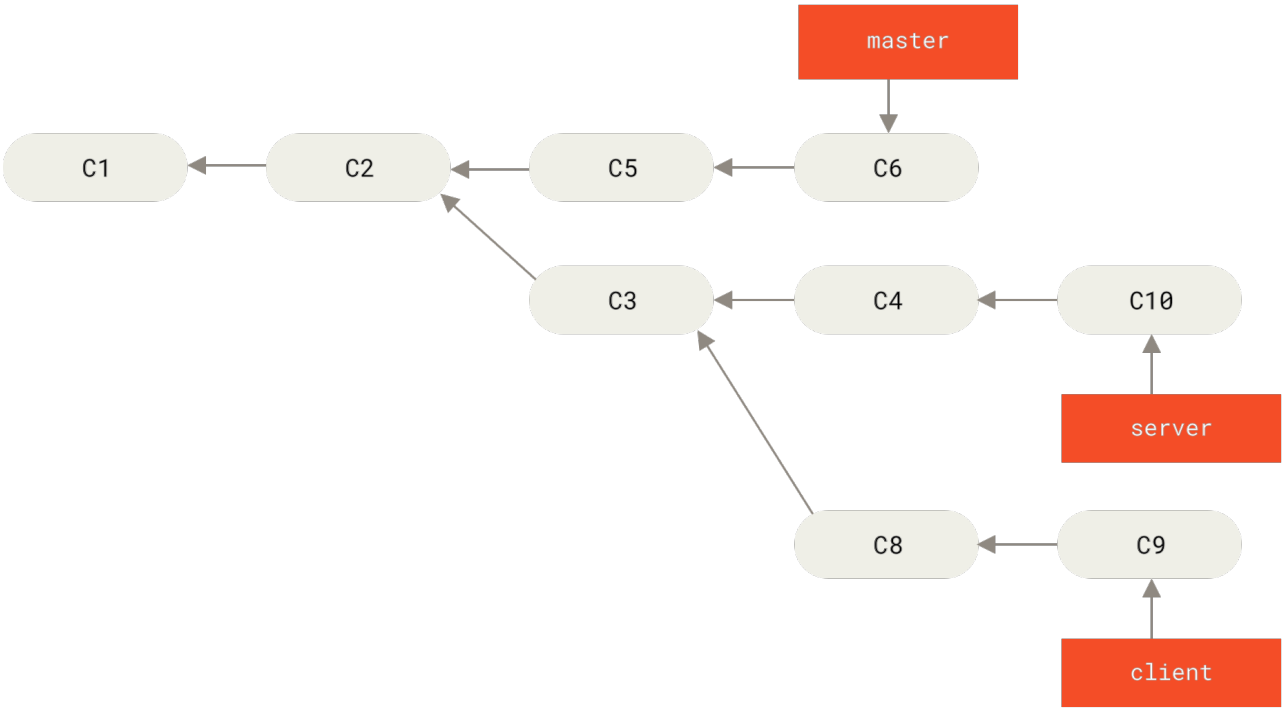

Figure 39. A history with a topic branch off another topic branch

Suppose you decide that you want to merge your client-side changes into your mainline for a release, but you want to hold off on the server-side changes until it's tested further. You can take the changes on `client` that aren't on `server` (`C8` and `C9`) and replay them on your `master` branch by using the `--onto` option of `git rebase`:

```
$ git rebase --onto master server client
```

This basically says, "Take the `client` branch, figure out the patches since it diverged from the `server` branch, and replay these patches in the `client` branch as if it was based directly off the `master` branch instead." It's a bit complex, but the result is pretty cool.

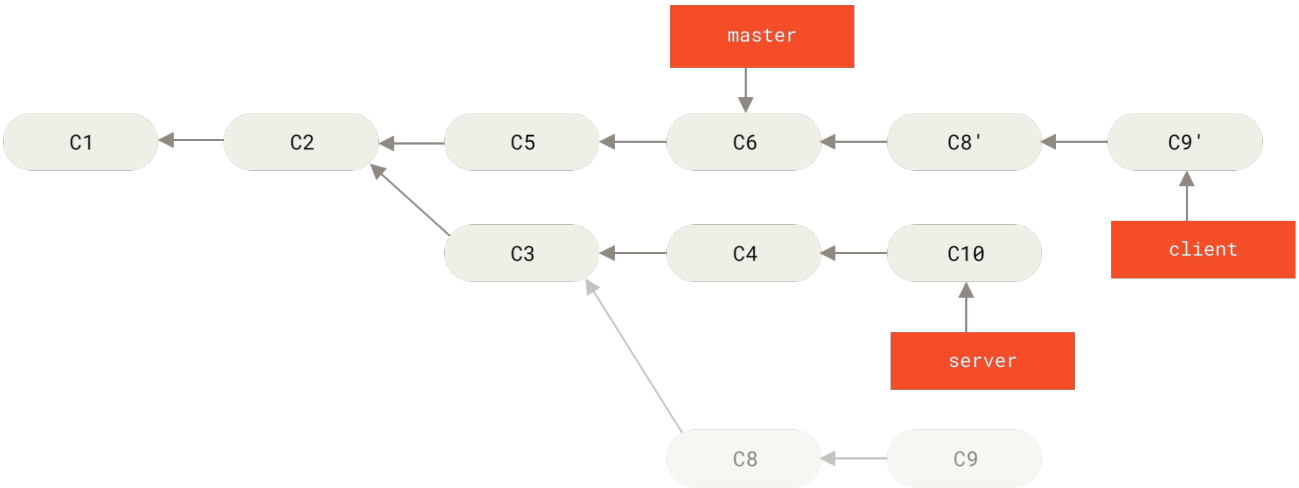

Figure 40. Rebasing a topic branch off another topic branch

Now you can fast-forward your `master` branch (see Fast-forwarding your `master` branch to include the `client` branch changes):

```
$ git checkout master
$ git merge client
```

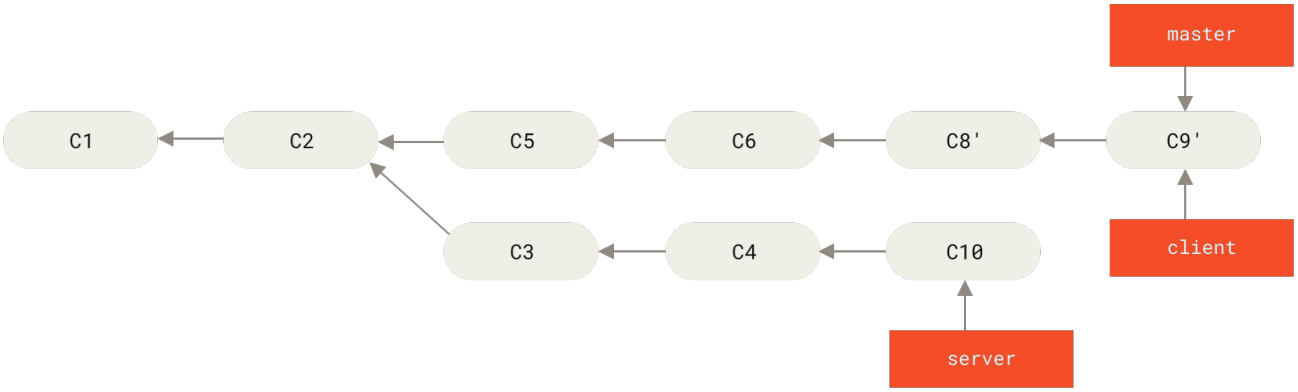

Figure 41. Fast-forwarding your `master` branch to include the `client` branch changes

Let's say you decide to pull in your `server` branch as well. You can rebase the `server` branch onto the `master` branch without having to check it out first by running `git rebase <basebranch> <topicbranch>` — which checks out the topic branch (in this case, `server`) for you and replays it onto the base branch (`master`):

```
$ git rebase master server
```

This replays your `server` work on top of your `master` work, as shown in Rebasing your `server` branch on top of your `master` branch.

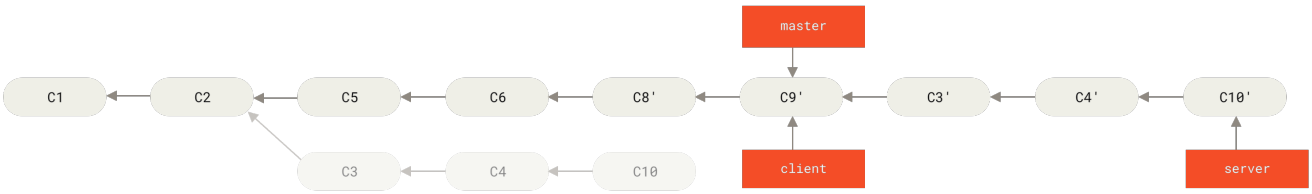

Figure 42. Rebasing your `server` branch on top of your `master` branch

Then, you can fast-forward the base branch (`master`):

```
$ git checkout master
$ git merge server
```

You can remove the `client` and `server` branches because all the work is integrated and you don't need them anymore, leaving your history for this entire process looking like Final commit history:

```
$ git branch -d client
$ git branch -d server
```

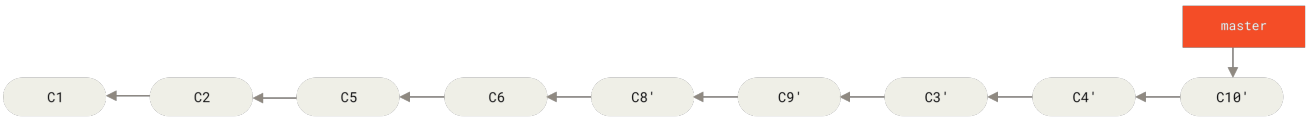

Figure 43. Final commit history

The Perils of Rebasing

Ahh, but the bliss of rebasing isn't without its drawbacks, which can be summed up in a single line:

Do not rebase commits that exist outside your repository and that people may have based work on.

If you follow that guideline, you'll be fine. If you don't, people will hate you, and you'll be scorned by friends and family.

When you rebase stuff, you're abandoning existing commits and creating new ones that are similar but different. If you push commits somewhere and others pull them down and base work on them, and then you rewrite those commits with `git rebase` and push them up again, your collaborators will have to re-merge their work and things will get messy when you try to pull their work back into yours.

Let's look at an example of how rebasing work that you've made public can cause problems. Suppose you clone from a central server and then do some work off that. Your commit history looks like this:

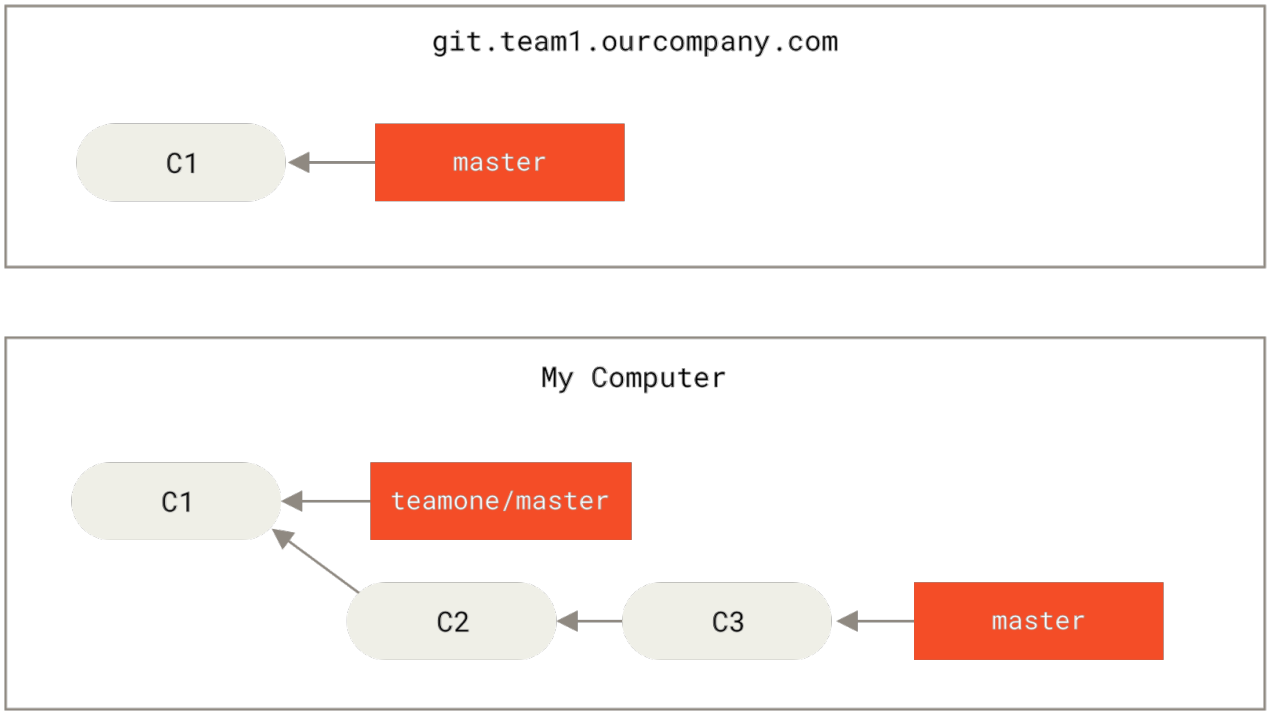

Figure 44. Clone a repository, and base some work on it

Now, someone else does more work that includes a merge, and pushes that work to the central server. You fetch it and merge the new remote branch into your work, making your history look something like this:

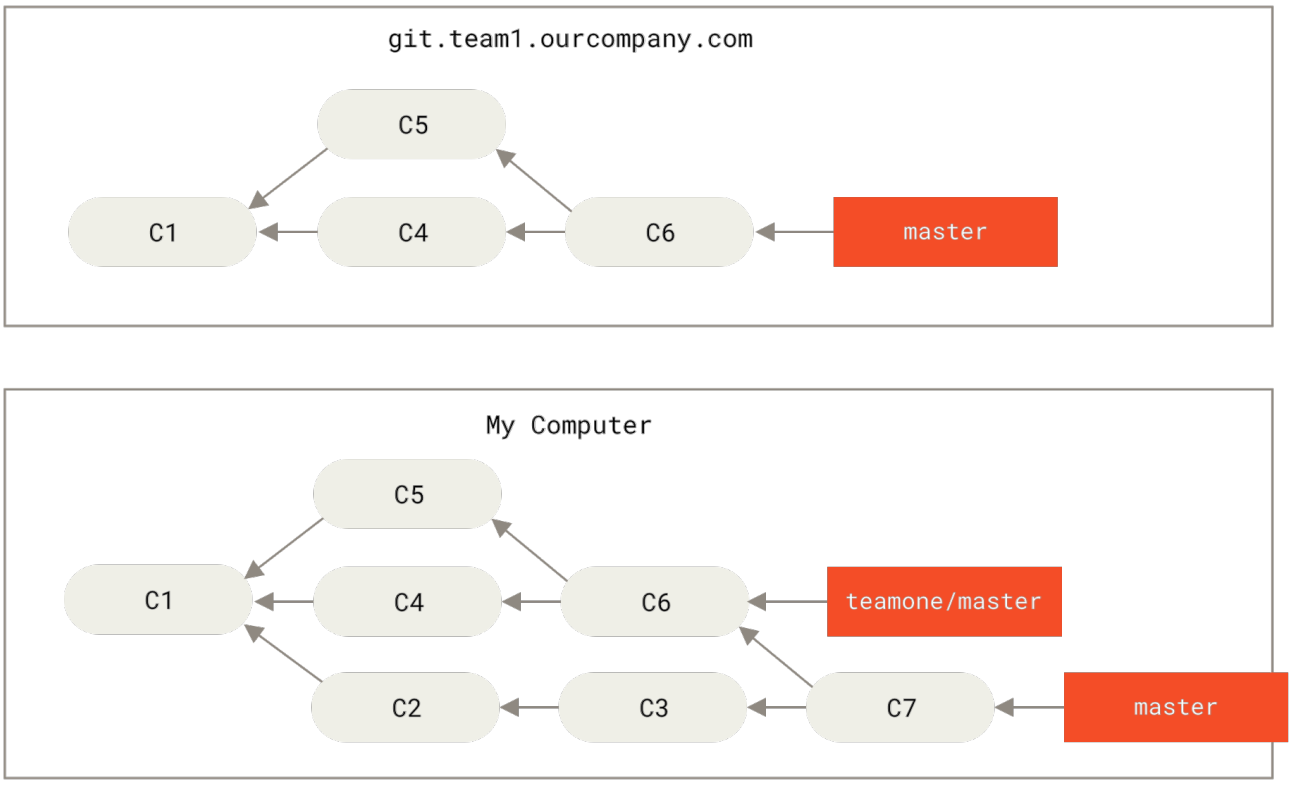

Figure 45. Fetch more commits, and merge them into your work

Next, the person who pushed the merged work decides to go back and rebase their work instead; they do a `git push --force` to overwrite the history on the server. You then fetch from that server, bringing down the new commits.

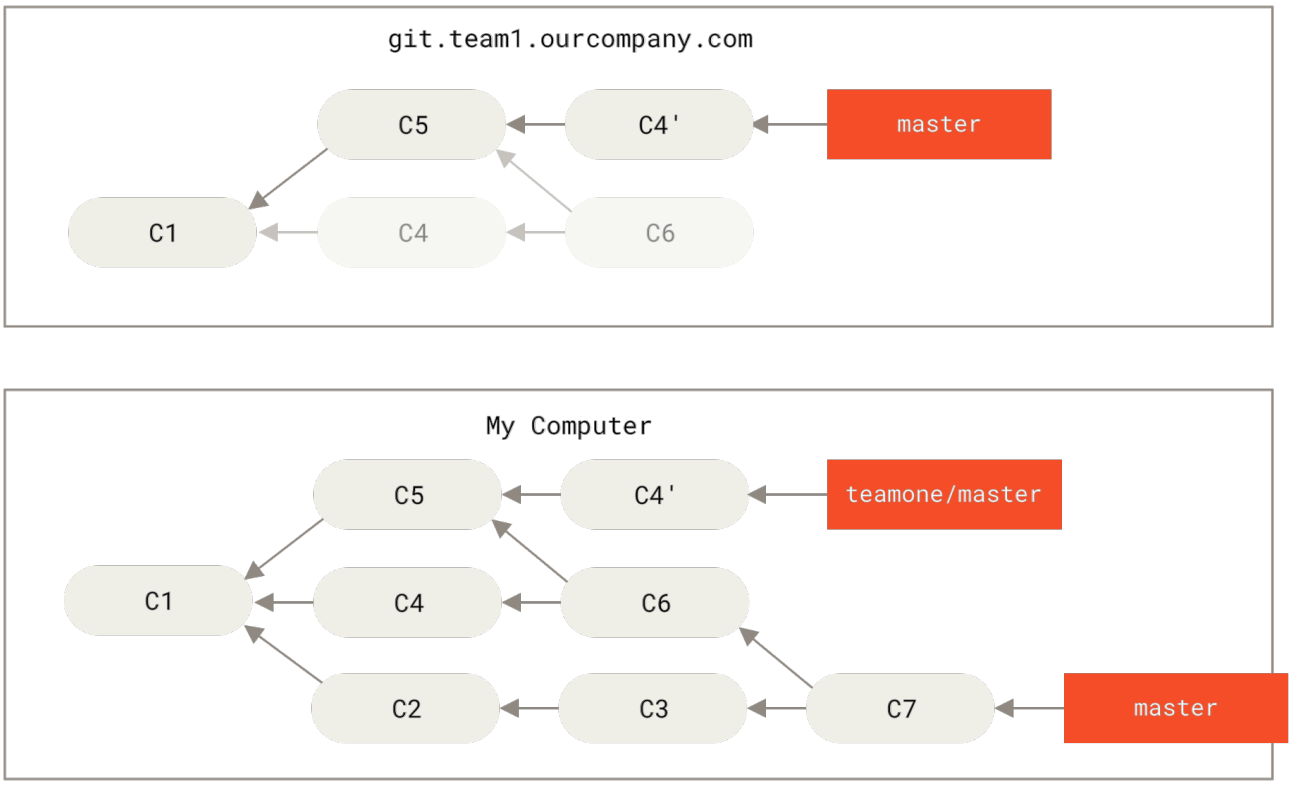

Figure 46. Someone pushes rebased commits, abandoning commits you've based your work on

Now you're both in a pickle. If you do a `git pull`, you'll create a merge commit which includes both lines of history, and your repository will look like this:

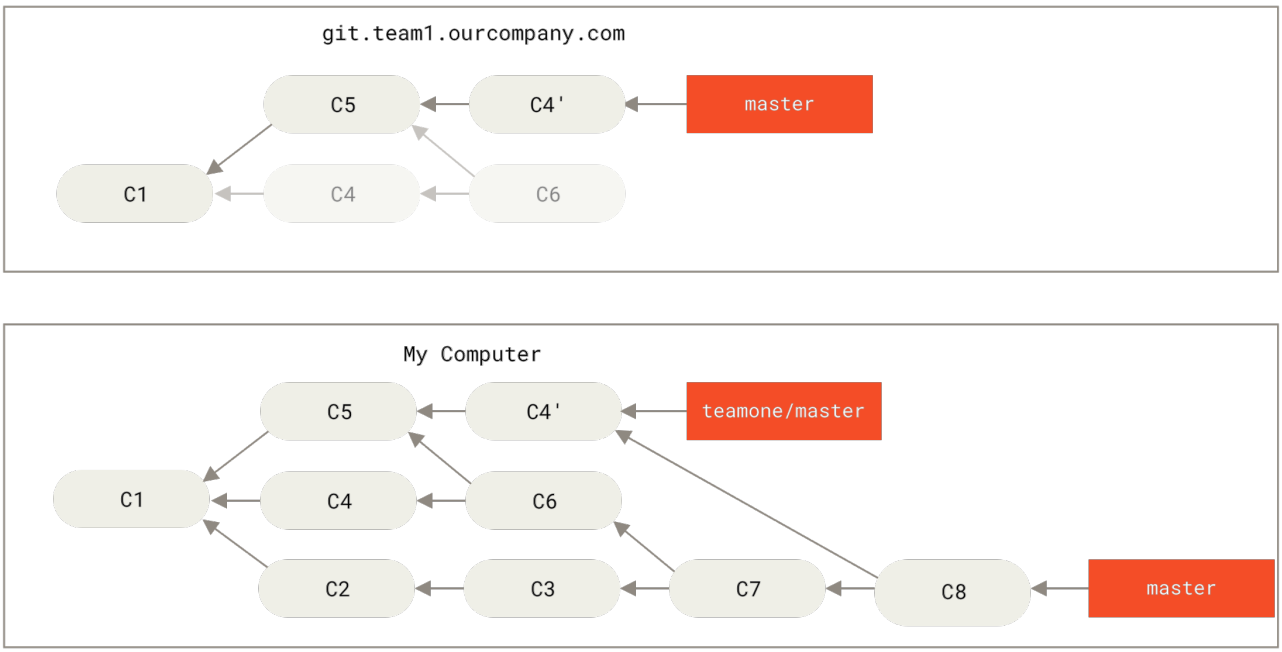

Figure 47. You merge in the same work again into a new merge commit

If you run a `git log` when your history looks like this, you'll see two commits that have the same author, date, and message, which will be confusing. Furthermore, if you push this history back up to the server, you'll reintroduce all those rebased commits to the central server, which can further confuse people. It's pretty safe to assume that the other developer doesn't want `C4` and `C6` to be in the history; that's why they rebased in the first place.

Rebase When You Rebase

If you **do** find yourself in a situation like this, Git has some further magic that might help you out. If someone on your team force pushes changes that overwrite work that you've based work on, your challenge is to figure out what is yours and what they've rewritten.

It turns out that in addition to the commit SHA-1 checksum, Git also calculates a checksum that is based just on the patch introduced with the commit. This is called a "patch-id".

If you pull down work that was rewritten and rebase it on top of the new commits from your partner, Git can often successfully figure out what is uniquely yours and apply them back on top of the new branch.

For instance, in the previous scenario, if instead of doing a merge when we're at Someone pushes rebased commits, abandoning commits you've based your work on we run `git rebase teamone/master`, Git will:

- Determine what work is unique to our branch (`C2`, `C3`, `C4`, `C6`, `C7`)
- Determine which are not merge commits (`C2`, `C3`, `C4`)
- Determine which have not been rewritten into the target branch (just `C2` and `C3`, since `C4` is the same patch as `C4'`)

- Apply those commits to the top of `teamone/master`

So instead of the result we see in You merge in the same work again into a new merge commit, we would end up with something more like Rebase on top of force-pushed rebase work.

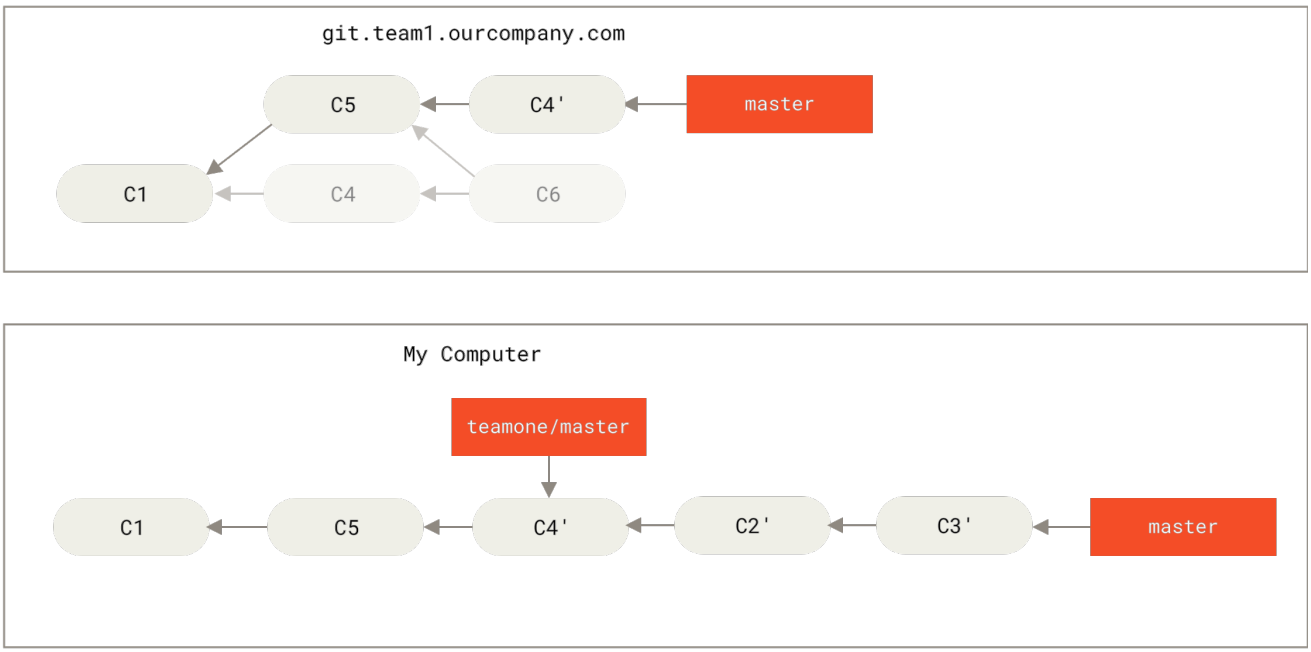

Figure 48. Rebase on top of force-pushed rebase work

This only works if `C4` and `C4'` that your partner made are almost exactly the same patch. Otherwise the rebase won't be able to tell that it's a duplicate and will add another `C4`-like patch (which will probably fail to apply cleanly, since the changes would already be at least somewhat there).

You can also simplify this by running a `git pull --rebase` instead of a normal `git pull`. Or you could do it manually with a `git fetch` followed by a `git rebase teamone/master` in this case.

If you are using `git pull` and want to make `--rebase` the default, you can set the `pull.rebase` config value with something like `git config --global pull.rebase true`.

If you only ever rebase commits that have never left your own computer, you'll be just fine. If you rebase commits that have been pushed, but that no one else has based commits from, you'll also be fine. If you rebase commits that have already been pushed publicly, and people may have based work on those commits, then you may be in for some frustrating trouble, and the scorn of your teammates.

If you or a partner does find it necessary at some point, make sure everyone knows to run `git pull --rebase` to try to make the pain after it happens a little bit simpler.

Rebase vs. Merge

Now that you've seen rebasing and merging in action, you may be wondering which one is better. Before we can answer this, let's step back a bit and talk about what history means.

One point of view on this is that your repository's commit history is a **record of what actually happened.** It's a historical document, valuable in its own right, and shouldn't be tampered with.

From this angle, changing the commit history is almost blasphemous; you're *lying* about what actually transpired. So what if there was a messy series of merge commits? That's how it happened, and the repository should preserve that for posterity.

The opposing point of view is that the commit history is the **story of how your project was made.** You wouldn't publish the first draft of a book, so why show your messy work? When you're working on a project, you may need a record of all your missteps and dead-end paths, but when it's time to show your work to the world, you may want to tell a more coherent story of how to get from A to B. People in this camp use tools like `rebase` and `filter-branch` to rewrite their commits before they're merged into the mainline branch. They use tools like `rebase` and `filter-branch`, to tell the story in the way that's best for future readers.

Now, to the question of whether merging or rebasing is better: hopefully you'll see that it's not that simple. Git is a powerful tool, and allows you to do many things to and with your history, but every team and every project is different. Now that you know how both of these things work, it's up to you to decide which one is best for your particular situation.

You can get the best of both worlds: rebase local changes before pushing to clean up your work, but never rebase anything that you've pushed somewhere.

Summary

We've covered basic branching and merging in Git. You should feel comfortable creating and switching to new branches, switching between branches and merging local branches together. You should also be able to share your branches by pushing them to a shared server, working with others on shared branches and rebasing your branches before they are shared. Next, we'll cover what you'll need to run your own Git repository-hosting server.

Git on the Server

At this point, you should be able to do most of the day-to-day tasks for which you'll be using Git. However, in order to do any collaboration in Git, you'll need to have a remote Git repository. Although you can technically push changes to and pull changes from individuals' repositories, doing so is discouraged because you can fairly easily confuse what they're working on if you're not careful. Furthermore, you want your collaborators to be able to access the repository even if your computer is offline — having a more reliable common repository is often useful. Therefore, the preferred method for collaborating with someone is to set up an intermediate repository that you both have access to, and push to and pull from that.

Running a Git server is fairly straightforward. First, you choose which protocols you want your server to support. The first section of this chapter will cover the available protocols and the pros and cons of each. The next sections will explain some typical setups using those protocols and how to get your server running with them. Last, we'll go over a few hosted options, if you don't mind hosting your code on someone else's server and don't want to go through the hassle of setting up and maintaining your own server.

If you have no interest in running your own server, you can skip to the last section of the chapter to see some options for setting up a hosted account and then move on to the next chapter, where we discuss the various ins and outs of working in a distributed source control environment.

A remote repository is generally a *bare repository* — a Git repository that has no working directory. Because the repository is only used as a collaboration point, there is no reason to have a snapshot checked out on disk; it's just the Git data. In the simplest terms, a bare repository is the contents of your project's `.git` directory and nothing else.

The Protocols

Git can use four distinct protocols to transfer data: Local, HTTP, Secure Shell (SSH) and Git. Here we'll discuss what they are and in what basic circumstances you would want (or not want) to use them.

Local Protocol

The most basic is the *Local protocol*, in which the remote repository is in another directory on the same host. This is often used if everyone on your team has access to a shared filesystem such as an NFS mount, or in the less likely case that everyone logs in to the same computer. The latter wouldn't be ideal, because all your code repository instances would reside on the same computer, making a catastrophic loss much more likely.

If you have a shared mounted filesystem, then you can clone, push to, and pull from a local file-based repository. To clone a repository like this, or to add one as a remote to an existing project, use the path to the repository as the URL. For example, to clone a local repository, you can run something like this:

```
$ git clone /srv/git/project.git
```

Or you can do this:

```
$ git clone file:///srv/git/project.git
```

Git operates slightly differently if you explicitly specify `file://` at the beginning of the URL. If you just specify the path, Git tries to use hardlinks or directly copy the files it needs. If you specify `file://`, Git fires up the processes that it normally uses to transfer data over a network, which is generally much less efficient. The main reason to specify the `file://` prefix is if you want a clean copy of the repository with extraneous references or objects left out — generally after an import from another VCS or something similar (see Git Internals for maintenance tasks). We'll use the normal path here because doing so is almost always faster.

To add a local repository to an existing Git project, you can run something like this:

```
$ git remote add local_proj /srv/git/project.git
```

Then, you can push to and pull from that remote via your new remote name `local_proj` as though you were doing so over a network.

The Pros

The pros of file-based repositories are that they're simple and they use existing file permissions and network access. If you already have a shared filesystem to which your whole team has access, setting up a repository is very easy. You stick the bare repository copy somewhere everyone has shared access to and set the read/write permissions as you would for any other shared directory. We'll discuss how to export a bare repository copy for this purpose in Getting Git on a Server.

This is also a nice option for quickly grabbing work from someone else's working repository. If you and a co-worker are working on the same project and they want you to check something out, running a command like `git pull /home/john/project` is often easier than them pushing to a remote server and you subsequently fetching from it.

The Cons

The cons of this method are that shared access is generally more difficult to set up and reach from multiple locations than basic network access. If you want to push from your laptop when you're at home, you have to mount the remote disk, which can be difficult and slow compared to network-based access.

It's important to mention that this isn't necessarily the fastest option if you're using a shared mount of some kind. A local repository is fast only if you have fast access to the data. A repository on NFS is often slower than the repository over SSH on the same server, allowing Git to run off local disks on each system.

Finally, this protocol does not protect the repository against accidental damage. Every user has full shell access to the "remote" directory, and there is nothing preventing them from changing or removing internal Git files and corrupting the repository.

The HTTP Protocols

Git can communicate over HTTP using two different modes. Prior to Git 1.6.6, there was only one way it could do this which was very simple and generally read-only. In version 1.6.6, a new, smarter protocol was introduced that involved Git being able to intelligently negotiate data transfer in a manner similar to how it does over SSH. In the last few years, this new HTTP protocol has become very popular since it's simpler for the user and smarter about how it communicates. The newer version is often referred to as the *Smart* HTTP protocol and the older way as *Dumb* HTTP. We'll cover the newer Smart HTTP protocol first.

Smart HTTP

Smart HTTP operates very similarly to the SSH or Git protocols but runs over standard HTTPS ports and can use various HTTP authentication mechanisms, meaning it's often easier on the user than something like SSH, since you can use things like username/password authentication rather than having to set up SSH keys.

It has probably become the most popular way to use Git now, since it can be set up to both serve anonymously like the `git://` protocol, and can also be pushed over with authentication and encryption like the SSH protocol. Instead of having to set up different URLs for these things, you can now use a single URL for both. If you try to push and the repository requires authentication (which it normally should), the server can prompt for a username and password. The same goes for read access.

In fact, for services like GitHub, the URL you use to view the repository online (for example, https://github.com/schacon/simplegit) is the same URL you can use to clone and, if you have access, push over.

Dumb HTTP

If the server does not respond with a Git HTTP smart service, the Git client will try to fall back to the simpler *Dumb* HTTP protocol. The Dumb protocol expects the bare Git repository to be served like normal files from the web server. The beauty of Dumb HTTP is the simplicity of setting it up. Basically, all you have to do is put a bare Git repository under your HTTP document root and set up a specific `post-update` hook, and you're done (see Git Hooks). At that point, anyone who can access the web server under which you put the repository can also clone your repository. To allow read access to your repository over HTTP, do something like this:

```
$ cd /var/www/htdocs/
$ git clone --bare /path/to/git_project gitproject.git
$ cd gitproject.git
$ mv hooks/post-update.sample hooks/post-update
$ chmod a+x hooks/post-update
```

That's all. The `post-update` hook that comes with Git by default runs the appropriate command (`git update-server-info`) to make HTTP fetching and cloning work properly. This command is run when you push to this repository (over SSH perhaps); then, other people can clone via something like:

```
$ git clone https://example.com/gitproject.git
```

In this particular case, we're using the `/var/www/htdocs` path that is common for Apache setups, but you can use any static web server — just put the bare repository in its path. The Git data is served as basic static files (see the Git Internals chapter for details about exactly how it's served).

Generally you would either choose to run a read/write Smart HTTP server or simply have the files accessible as read-only in the Dumb manner. It's rare to run a mix of the two services.

The Pros

We'll concentrate on the pros of the Smart version of the HTTP protocol.

The simplicity of having a single URL for all types of access and having the server prompt only when authentication is needed makes things very easy for the end user. Being able to authenticate with a username and password is also a big advantage over SSH, since users don't have to generate SSH keys locally and upload their public key to the server before being able to interact with it. For less sophisticated users, or users on systems where SSH is less common, this is a major advantage in usability. It is also a very fast and efficient protocol, similar to the SSH one.

You can also serve your repositories read-only over HTTPS, which means you can encrypt the content transfer; or you can go so far as to make the clients use specific signed SSL certificates.

Another nice thing is that HTTP and HTTPS are such commonly used protocols that corporate firewalls are often set up to allow traffic through their ports.

The Cons

Git over HTTPS can be a little more tricky to set up compared to SSH on some servers. Other than that, there is very little advantage that other protocols have over Smart HTTP for serving Git content.

If you're using HTTP for authenticated pushing, providing your credentials is sometimes more complicated than using keys over SSH. There are, however, several credential caching tools you can use, including Keychain access on macOS and Credential Manager on Windows, to make this pretty painless. Read Credential Storage to see how to set up secure HTTP password caching on your system.

The SSH Protocol

A common transport protocol for Git when self-hosting is over SSH. This is because SSH access to servers is already set up in most places — and if it isn't, it's easy to do. SSH is also an authenticated network protocol and, because it's ubiquitous, it's generally easy to set up and use.

To clone a Git repository over SSH, you can specify an `ssh://` URL like this:

```
$ git clone ssh://[user@]server/project.git
```

Or you can use the shorter scp-like syntax for the SSH protocol:

```
$ git clone [user@]server:project.git
```

In both cases above, if you don't specify the optional username, Git assumes the user you're currently logged in as.

The Pros

The pros of using SSH are many. First, SSH is relatively easy to set up — SSH daemons are commonplace, many network admins have experience with them, and many OS distributions are set up with them or have tools to manage them. Next, access over SSH is secure — all data transfer is encrypted and authenticated. Last, like the HTTPS, Git and Local protocols, SSH is efficient, making the data as compact as possible before transferring it.

The Cons

The negative aspect of SSH is that it doesn't support anonymous access to your Git repository. If you're using SSH, people *must* have SSH access to your machine, even in a read-only capacity, which doesn't make SSH conducive to open source projects for which people might simply want to clone your repository to examine it. If you're using it only within your corporate network, SSH may be the only protocol you need to deal with. If you want to allow anonymous read-only access to your projects and also want to use SSH, you'll have to set up SSH for you to push over but something else for others to fetch from.

The Git Protocol

Finally, we have the Git protocol. This is a special daemon that comes packaged with Git; it listens on a dedicated port (9418) that provides a service similar to the SSH protocol, but with absolutely no authentication or cryptography. In order for a repository to be served over the Git protocol, you must create a `git-daemon-export-ok` file — the daemon won't serve a repository without that file in it — but, other than that, there is no security. Either the Git repository is available for everyone to clone, or it isn't. This means that there is generally no pushing over this protocol. You can enable push access but, given the lack of authentication, anyone on the internet who finds your project's URL could push to that project. Suffice it to say that this is rare.

The Pros

The Git protocol is often the fastest network transfer protocol available. If you're serving a lot of traffic for a public project or serving a very large project that doesn't require user authentication for read access, it's likely that you'll want to set up a Git daemon to serve your project. It uses the same data-transfer mechanism as the SSH protocol but without the encryption and authentication overhead.

The Cons

Due to the lack of TLS or other cryptography, cloning over `git://` might lead to an arbitrary code execution vulnerability, and should therefore be avoided unless you know what you are doing.

- If you run `git clone git://example.com/project.git`, an attacker who controls e.g your router can modify the repo you just cloned, inserting malicious code into it. If you then compile/run the code you just cloned, you will execute the malicious code. Running `git clone http://example.com/project.git` should be avoided for the same reason.
- Running `git clone https://example.com/project.git` does not suffer from the same problem (unless the attacker can provide a TLS certificate for example.com). Running `git clone git@example.com:project.git` only suffers from this problem if you accept a wrong SSH key fingerprint.

It also has no authentication, i.e. anyone can clone the repo (although this is often exactly what you want). It's also probably the most difficult protocol to set up. It must run its own daemon, which requires `xinetd` or `systemd` configuration or the like, which isn't always a walk in the park. It also requires firewall access to port 9418, which isn't a standard port that corporate firewalls always allow. Behind big corporate firewalls, this obscure port is commonly blocked.

Getting Git on a Server

Now we'll cover setting up a Git service running these protocols on your own server.

Here we'll be demonstrating the commands and steps needed to do basic, simplified installations on a Linux-based server, though it's also possible to run these services on macOS or Windows servers. Actually setting up a production server within your infrastructure will certainly entail differences in security measures or operating system tools, but hopefully this will give you the general idea of what's involved.

In order to initially set up any Git server, you have to export an existing repository into a new bare repository — a repository that doesn't contain a working directory. This is generally straightforward to do. In order to clone your repository to create a new bare repository, you run the clone command with the `--bare` option. By convention, bare repository directory names end with the suffix `.git`, like so:

```
$ git clone --bare my_project my_project.git
Cloning into bare repository 'my_project.git'...
done.
```

You should now have a copy of the Git directory data in your `my_project.git` directory.

This is roughly equivalent to something like:

```
$ cp -Rf my_project/.git my_project.git
```

There are a couple of minor differences in the configuration file but, for your purpose, this is close to the same thing. It takes the Git repository by itself, without a working directory, and creates a directory specifically for it alone.

Putting the Bare Repository on a Server

Now that you have a bare copy of your repository, all you need to do is put it on a server and set up your protocols. Let's say you've set up a server called `git.example.com` to which you have SSH access, and you want to store all your Git repositories under the `/srv/git` directory. Assuming that `/srv/git` exists on that server, you can set up your new repository by copying your bare repository over:

```
$ scp -r my_project.git user@git.example.com:/srv/git
```

At this point, other users who have SSH-based read access to the `/srv/git` directory on that server can clone your repository by running:

```
$ git clone user@git.example.com:/srv/git/my_project.git
```

If a user SSHs into a server and has write access to the `/srv/git/my_project.git` directory, they will also automatically have push access.

Git will automatically add group write permissions to a repository properly if you run the `git init` command with the `--shared` option. Note that by running this command, you will not destroy any commits, refs, etc. in the process.

```
$ ssh user@git.example.com
$ cd /srv/git/my_project.git
$ git init --bare --shared
```

You see how easy it is to take a Git repository, create a bare version, and place it on a server to which you and your collaborators have SSH access. Now you're ready to collaborate on the same project.

It's important to note that this is literally all you need to do to run a useful Git server to which several people have access — just add SSH-able accounts on a server, and stick a bare repository somewhere that all those users have read and write access to. You're ready to go — nothing else needed.

In the next few sections, you'll see how to expand to more sophisticated setups. This discussion will include not having to create user accounts for each user, adding public read access to repositories, setting up web UIs and more. However, keep in mind that to collaborate with a couple of people on a private project, all you *need* is an SSH server and a bare repository.

Small Setups

If you're a small outfit or are just trying out Git in your organization and have only a few developers, things can be simple for you. One of the most complicated aspects of setting up a Git server is user management. If you want some repositories to be read-only for certain users and read/write for others, access and permissions can be a bit more difficult to arrange.

SSH Access

If you have a server to which all your developers already have SSH access, it's generally easiest to set up your first repository there, because you have to do almost no work (as we covered in the last section). If you want more complex access control type permissions on your repositories, you can handle them with the normal filesystem permissions of your server's operating system.

If you want to place your repositories on a server that doesn't have accounts for everyone on your team for whom you want to grant write access, then you must set up SSH access for them. We assume that if you have a server with which to do this, you already have an SSH server installed, and that's how you're accessing the server.

There are a few ways you can give access to everyone on your team. The first is to set up accounts for everybody, which is straightforward but can be cumbersome. You may not want to run `adduser` (or the possible alternative `useradd`) and have to set temporary passwords for every new user.

A second method is to create a single 'git' user account on the machine, ask every user who is to have write access to send you an SSH public key, and add that key to the `~/.ssh/authorized_keys` file of that new 'git' account. At that point, everyone will be able to access that machine via the 'git' account. This doesn't affect the commit data in any way — the SSH user you connect as doesn't affect the commits you've recorded.

Another way to do it is to have your SSH server authenticate from an LDAP server or some other centralized authentication source that you may already have set up. As long as each user can get shell access on the machine, any SSH authentication mechanism you can think of should work.

Generating Your SSH Public Key

Many Git servers authenticate using SSH public keys. In order to provide a public key, each user in your system must generate one if they don't already have one. This process is similar across all operating systems. First, you should check to make sure you don't already have a key. By default, a user's SSH keys are stored in that user's `~/.ssh` directory. You can easily check to see if you have a key already by going to that directory and listing the contents:

```
$ cd ~/.ssh
$ ls
authorized_keys2  id_dsa       known_hosts
config            id_dsa.pub
```

You're looking for a pair of files named something like `id_dsa` or `id_rsa` and a matching file with a `.pub` extension. The `.pub` file is your public key, and the other file is the corresponding private key. If you don't have these files (or you don't even have a `.ssh` directory), you can create them by running a program called `ssh-keygen`, which is provided with the SSH package on Linux/macOS systems and comes with Git for Windows:

```
$ ssh-keygen -o
Generating public/private rsa key pair.
Enter file in which to save the key (/home/schacon/.ssh/id_rsa):
```

```
Created directory '/home/schacon/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/schacon/.ssh/id_rsa.
Your public key has been saved in /home/schacon/.ssh/id_rsa.pub.
The key fingerprint is:
d0:82:24:8e:d7:f1:bb:9b:33:53:96:93:49:da:9b:e3 schacon@mylaptop.local
```

First it confirms where you want to save the key (`.ssh/id_rsa`), and then it asks twice for a passphrase, which you can leave empty if you don't want to type a password when you use the key. However, if you do use a password, make sure to add the `-o` option; it saves the private key in a format that is more resistant to brute-force password cracking than is the default format. You can also use the `ssh-agent` tool to prevent having to enter the password each time.

Now, each user that does this has to send their public key to you or whoever is administrating the Git server (assuming you're using an SSH server setup that requires public keys). All they have to do is copy the contents of the `.pub` file and email it. The public keys look something like this:

```
$ cat ~/.ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEAklOUpkDHrfHY17SbrmTIpNLTGK9Tjom/BWDSU
GPl+nafzlHDTYW7hdI4yZ5ew18JH4JW9jbhUFrviQzM7xlELEVf4h9lFX5QVkbPppSwg0cda3
Pbv7kOdJ/MTyBlWXFCR+HAo3FXRitBqxiX1nKhXpHAZsMciLq8V6RjsNAQwdsdMFvSlVK/7XA
t3FaoJoAsncM1Q9x5+3V0Ww68/eIFmb1zuUFljQJKprrX88XypNDvjYNby6vw/Pb0rwert/En
mZ+AW4OZPnTPI89ZPmVMLuayrD2cE86Z/il8b+gw3r3+1nKatmIkjn2so1d01QraTlMqVSsbx
NrRFi9wrf+M7Q== schacon@mylaptop.local
```

For a more in-depth tutorial on creating an SSH key on multiple operating systems, see the GitHub guide on SSH keys at https://docs.github.com/en/authentication/connecting-to-github-with-ssh/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent.

Setting Up the Server

Let's walk through setting up SSH access on the server side. In this example, you'll use the `authorized_keys` method for authenticating your users. We also assume you're running a standard Linux distribution like Ubuntu.

A good deal of what is described here can be automated by using the `ssh-copy-id` command, rather than manually copying and installing public keys.

First, you create a `git` user account and a `.ssh` directory for that user.

```
$ sudo adduser git
$ su git
$ cd
$ mkdir .ssh && chmod 700 .ssh
$ touch .ssh/authorized_keys && chmod 600 .ssh/authorized_keys
```

Next, you need to add some developer SSH public keys to the `authorized_keys` file for the `git` user. Let's assume you have some trusted public keys and have saved them to temporary files. Again, the public keys look something like this:

```
$ cat /tmp/id_rsa.john.pub
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQCB007n/ww+ouN4gSLKssMxXnBOvf9LGt4L
ojG6rs6hPB09j9R/T17/x4lhJA0F3FR1rP6kYBRsWj2aThGw6HXLm9/5zytK6Ztg3RPKK+4k
Yjh6541NYsnEAZuXz0jTTyAUfrtU3Z5E003C4oxOj6H0rfIF1kKI9MAQLMdpGW1GYEIgS9Ez
Sdfd8AcCIicTDWbqLAcU4UpkaX8KyGlLwsNuuGztobF8m72ALC/nLF6JLtPofwFBlgc+myiv
O7TCUSBdLQlgMVOFq1I2uPWQOkOWQAHukEOmfjy2jctxSDBQ220ymjaNsHT4kgtZg2AYYgPq
dAv8JggJICUvax2T9va5 gsg-keypair
```

You just append them to the `git` user's `authorized_keys` file in its `.ssh` directory:

```
$ cat /tmp/id_rsa.john.pub >> ~/.ssh/authorized_keys
$ cat /tmp/id_rsa.josie.pub >> ~/.ssh/authorized_keys
$ cat /tmp/id_rsa.jessica.pub >> ~/.ssh/authorized_keys
```

Now, you can set up an empty repository for them by running `git init` with the `--bare` option, which initializes the repository without a working directory:

```
$ cd /srv/git
$ mkdir project.git
$ cd project.git
$ git init --bare
Initialized empty Git repository in /srv/git/project.git/
```

Then, John, Josie, or Jessica can push the first version of their project into that repository by adding it as a remote and pushing up a branch. Note that someone must shell onto the machine and create a bare repository every time you want to add a project. Let's use `gitserver` as the hostname of the server on which you've set up your `git` user and repository. If you're running it internally, and you set up DNS for `gitserver` to point to that server, then you can use the commands pretty much as is (assuming that `myproject` is an existing project with files in it):

```
# on John's computer
$ cd myproject
$ git init
$ git add .
$ git commit -m 'Initial commit'
$ git remote add origin git@gitserver:/srv/git/project.git
$ git push origin master
```

At this point, the others can clone it down and push changes back up just as easily:

```
$ git clone git@gitserver:/srv/git/project.git
```

```
$ cd project
$ vim README
$ git commit -am 'Fix for README file'
$ git push origin master
```

With this method, you can quickly get a read/write Git server up and running for a handful of developers.

You should note that currently all these users can also log into the server and get a shell as the `git` user. If you want to restrict that, you will have to change the shell to something else in the `/etc/passwd` file.

You can easily restrict the `git` user account to only Git-related activities with a limited shell tool called `git-shell` that comes with Git. If you set this as the `git` user account's login shell, then that account can't have normal shell access to your server. To use this, specify `git-shell` instead of `bash` or `csh` for that account's login shell. To do so, you must first add the full pathname of the `git-shell` command to `/etc/shells` if it's not already there:

```
$ cat /etc/shells   # see if git-shell is already in there. If not...
$ which git-shell   # make sure git-shell is installed on your system.
$ sudo -e /etc/shells  # and add the path to git-shell from last command
```

Now you can edit the shell for a user using `chsh <username> -s <shell>`:

```
$ sudo chsh git -s $(which git-shell)
```

Now, the `git` user can still use the SSH connection to push and pull Git repositories but can't shell onto the machine. If you try, you'll see a login rejection like this:

```
$ ssh git@gitserver
fatal: Interactive git shell is not enabled.
hint: ~/git-shell-commands should exist and have read and execute access.
Connection to gitserver closed.
```

At this point, users are still able to use SSH port forwarding to access any host the git server is able to reach. If you want to prevent that, you can edit the `authorized_keys` file and prepend the following options to each key you'd like to restrict:

```
no-port-forwarding,no-X11-forwarding,no-agent-forwarding,no-pty
```

The result should look like this:

```
$ cat ~/.ssh/authorized_keys
no-port-forwarding,no-X11-forwarding,no-agent-forwarding,no-pty ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAABAQCB007n/ww+ouN4gSLKssMxXnBOvf9LGt4LojG6rs6h
```

```
PB09j9R/T17/x4lhJA0F3FR1rP6kYBRsWj2aThGw6HXLm9/5zytK6Ztg3RPKK+4kYjh6541N
YsnEAZuXz0jTTyAUfrtU3Z5E003C4oxOj6H0rfIF1kKI9MAQLMdpGW1GYEIgS9EzSdfd8AcC
IicTDWbqLAcU4UpkaX8KyGlLwsNuuGztobF8m72ALC/nLF6JLtPofwFBlgc+myivO7TCUSBd
LQlgMVOFq1I2uPWQOkOWQAHukEOmfjy2jctxSDBQ220ymjaNsHT4kgtZg2AYYgPqdAv8JggJ
ICUvax2T9va5 gsg-keypair

no-port-forwarding,no-X11-forwarding,no-agent-forwarding,no-pty ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAABAQDEwENNMomTboYI+LJieaAY16qiXiH3wuvENhBG...
```

Now Git network commands will still work just fine but the users won't be able to get a shell. As the output states, you can also set up a directory in the `git` user's home directory that customizes the `git-shell` command a bit. For instance, you can restrict the Git commands that the server will accept or you can customize the message that users see if they try to SSH in like that. Run `git help shell` for more information on customizing the shell.

Git Daemon

Next we'll set up a daemon serving repositories using the "Git" protocol. This is a common choice for fast, unauthenticated access to your Git data. Remember that since this is not an authenticated service, anything you serve over this protocol is public within its network.

If you're running this on a server outside your firewall, it should be used only for projects that are publicly visible to the world. If the server you're running it on is inside your firewall, you might use it for projects that a large number of people or computers (continuous integration or build servers) have read-only access to, when you don't want to have to add an SSH key for each.

In any case, the Git protocol is relatively easy to set up. Basically, you need to run this command in a daemonized manner:

```
$ git daemon --reuseaddr --base-path=/srv/git/ /srv/git/
```

The `--reuseaddr` option allows the server to restart without waiting for old connections to time out, while the `--base-path` option allows people to clone projects without specifying the entire path, and the path at the end tells the Git daemon where to look for repositories to export. If you're running a firewall, you'll also need to punch a hole in it at port 9418 on the box you're setting this up on.

You can daemonize this process a number of ways, depending on the operating system you're running.

Since `systemd` is the most common init system among modern Linux distributions, you can use it for that purpose. Simply place a file in `/etc/systemd/system/git-daemon.service` with these contents:

```
[Unit]
Description=Start Git Daemon

[Service]
ExecStart=/usr/bin/git daemon --reuseaddr --base-path=/srv/git/ /srv/git/
```

```
Restart=always
RestartSec=500ms

StandardOutput=syslog
StandardError=syslog
SyslogIdentifier=git-daemon

User=git
Group=git

[Install]
WantedBy=multi-user.target
```

You might have noticed that Git daemon is started here with `git` as both group and user. Modify it to fit your needs and make sure the provided user exists on the system. Also, check that the Git binary is indeed located at `/usr/bin/git` and change the path if necessary.

Finally, you'll run `systemctl enable git-daemon` to automatically start the service on boot, and can start and stop the service with, respectively, `systemctl start git-daemon` and `systemctl stop git-daemon`.

On other systems, you may want to use `xinetd`, a script in your `sysvinit` system, or something else — as long as you get that command daemonized and watched somehow.

Next, you have to tell Git which repositories to allow unauthenticated Git server-based access to. You can do this in each repository by creating a file named `git-daemon-export-ok`.

```
$ cd /path/to/project.git
$ touch git-daemon-export-ok
```

The presence of that file tells Git that it's OK to serve this project without authentication.

Smart HTTP

We now have authenticated access through SSH and unauthenticated access through `git://`, but there is also a protocol that can do both at the same time. Setting up Smart HTTP is basically just enabling a CGI script that is provided with Git called `git-http-backend` on the server. This CGI will read the path and headers sent by a `git fetch` or `git push` to an HTTP URL and determine if the client can communicate over HTTP (which is true for any client since version 1.6.6). If the CGI sees that the client is smart, it will communicate smartly with it; otherwise it will fall back to the dumb behavior (so it is backward compatible for reads with older clients).

Let's walk through a very basic setup. We'll set this up with Apache as the CGI server. If you don't have Apache setup, you can do so on a Linux box with something like this:

```
$ sudo apt-get install apache2 apache2-utils
```

```
$ a2enmod cgi alias env
```

This also enables the `mod_cgi`, `mod_alias`, and `mod_env` modules, which are all needed for this to work properly.

You'll also need to set the Unix user group of the `/srv/git` directories to `www-data` so your web server can read- and write-access the repositories, because the Apache instance running the CGI script will (by default) be running as that user:

```
$ chgrp -R www-data /srv/git
```

Next we need to add some things to the Apache configuration to run the `git-http-backend` as the handler for anything coming into the `/git` path of your web server.

```
SetEnv GIT_PROJECT_ROOT /srv/git
SetEnv GIT_HTTP_EXPORT_ALL
ScriptAlias /git/ /usr/lib/git-core/git-http-backend/
```

If you leave out `GIT_HTTP_EXPORT_ALL` environment variable, then Git will only serve to unauthenticated clients the repositories with the `git-daemon-export-ok` file in them, just like the Git daemon did.

Finally you'll want to tell Apache to allow requests to `git-http-backend` and make writes be authenticated somehow, possibly with an Auth block like this:

```
<Files "git-http-backend">
    AuthType Basic
    AuthName "Git Access"
    AuthUserFile /srv/git/.htpasswd
    Require expr !(%{QUERY_STRING} -strmatch '*service=git-receive-pack*' ||
%{REQUEST_URI} =~ m#/git-receive-pack$#)
    Require valid-user
</Files>
```

That will require you to create a `.htpasswd` file containing the passwords of all the valid users. Here is an example of adding a "schacon" user to the file:

```
$ htpasswd -c /srv/git/.htpasswd schacon
```

There are tons of ways to have Apache authenticate users, you'll have to choose and implement one of them. This is just the simplest example we could come up with. You'll also almost certainly want to set this up over SSL so all this data is encrypted.

We don't want to go too far down the rabbit hole of Apache configuration specifics, since you could well be using a different server or have different authentication needs. The idea is that Git comes

with a CGI called `git-http-backend` that when invoked will do all the negotiation to send and receive data over HTTP. It does not implement any authentication itself, but that can easily be controlled at the layer of the web server that invokes it. You can do this with nearly any CGI-capable web server, so go with the one that you know best.

For more information on configuring authentication in Apache, check out the Apache docs here: https://httpd.apache.org/docs/current/howto/auth.html.

GitWeb

Now that you have basic read/write and read-only access to your project, you may want to set up a simple web-based visualizer. Git comes with a CGI script called GitWeb that is sometimes used for this.

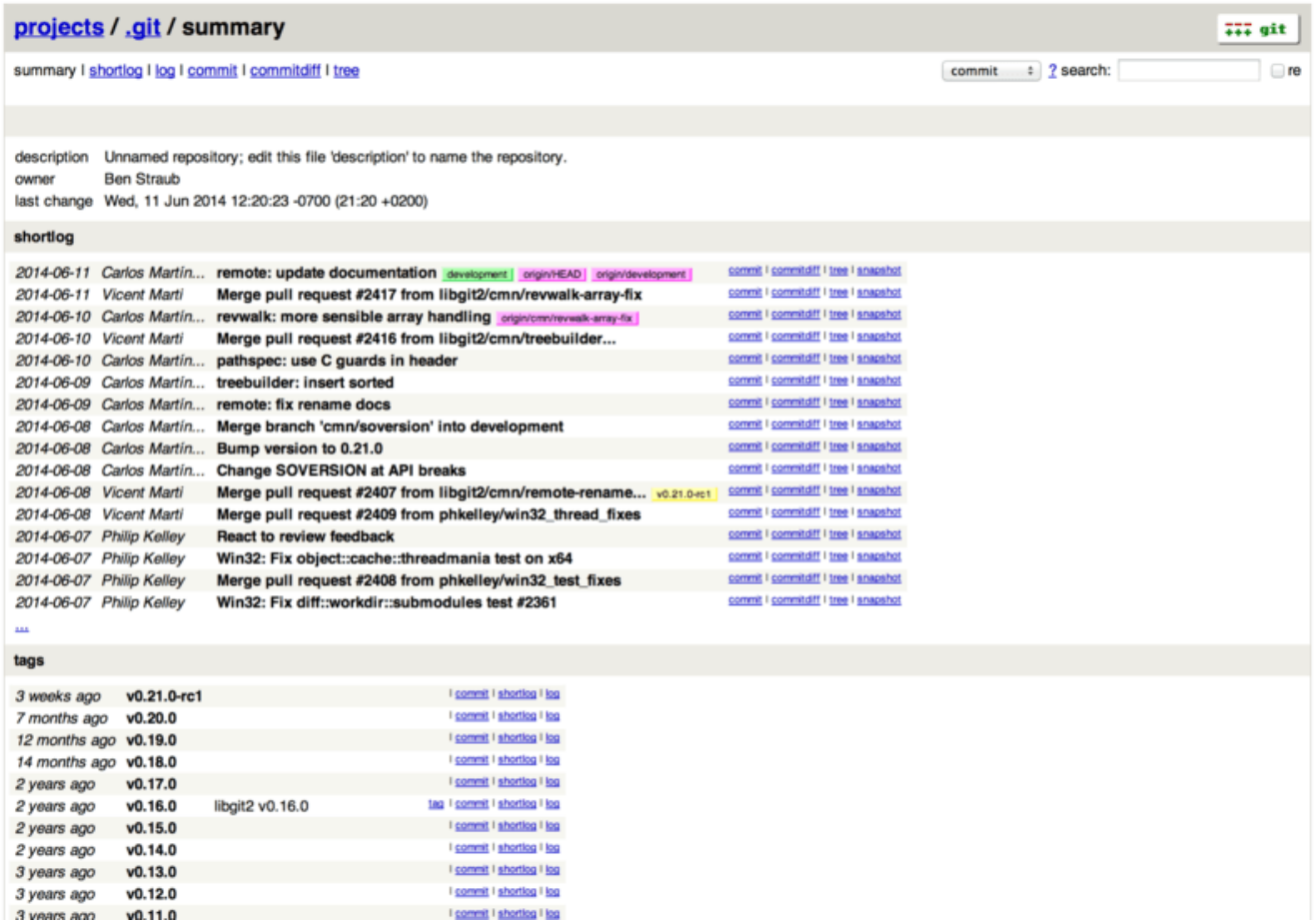

Figure 49. The GitWeb web-based user interface

If you want to check out what GitWeb would look like for your project, Git comes with a command to fire up a temporary instance if you have a lightweight web server on your system like `lighttpd` or `webrick`. On Linux machines, `lighttpd` is often installed, so you may be able to get it to run by typing `git instaweb` in your project directory. If you're running macOS, Leopard comes preinstalled with Ruby, so `webrick` may be your best bet. To start `instaweb` with a non-lighttpd handler, you can run it with the `--httpd` option.

```
$ git instaweb --httpd=webrick
[2009-02-21 10:02:21] INFO  WEBrick 1.3.1
```