

THE EXPERT'S VOICE®

SECOND EDITION

Pro Git

*EVERYTHING YOU NEED TO
KNOW ABOUT GIT*

Scott Chacon and Ben Straub

Apress®

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-sa/3.0/> or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

Prefacio por Scott Chacon

Bienvenidos a la segunda edición de Pro Git. La primera edición fue publicada hace más de cuatro años. Desde entonces mucho ha cambiado aunque muchas cosas importantes no. Mientras que la mayoría de los conceptos y comandos básicos siguen siendo válidos hoy gracias a que el equipo principal de Git es bastante fantástico manteniendo la compatibilidad con versiones anteriores, ha habido algunas adiciones y cambios significativos en la comunidad circundante a Git. La segunda edición de este libro pretende dar respuesta a esos cambios y actualizar el libro para que pueda ser más útil al nuevo usuario.

Cuando escribí la primera edición, Git seguía siendo relativamente difícil de usar y una herramienta escasamente utilizada por algunos hackers. Estaba empezando a cobrar fuerza en algunas comunidades, pero no había alcanzado la ubicuidad que tiene actualmente. Desde entonces, casi todas las comunidades de código abierto lo han adoptado. Git ha hecho un progreso increíble en Windows, en la explosión de interfaces gráficas de usuario para el mismo en todas las plataformas, en soporte IDE y en uso en empresas. El Pro Git de hace cuatro años no trataba nada de eso. Uno de los principales objetivos de esta nueva edición es tocar todas esas nuevas fronteras en la comunidad de Git.

La comunidad de código abierto que usa Git también se ha disparado. Cuando originalmente me pusé a escribir el libro hace casi cinco años (me tomó algún tiempo sacar la primera versión), acababa de empezar a trabajar en una empresa muy poco conocida desarrollando un sitio web de alojamiento Git llamada GitHub. En el momento de la publicación había quizás unos pocos miles de personas que utilizaban el sitio y sólo cuatro de nosotros trabajando en él. Al momento de escribir esta introducción, GitHub está anunciando nuestro proyecto alojado número 10 millones, con casi 5 millones de cuentas de desarrollador registradas y más de 230 empleados. Amado u odiado, GitHub ha cambiado en gran medida grandes franjas de la comunidad de código abierto de una manera que era apenas concebible cuando me senté a escribir la primera edición.

Escribí una pequeña sección en la versión original de Pro Git sobre GitHub como un ejemplo de Git hospedado con la cual nunca me sentí muy cómodo. No me gustaba estar escribiendo sobre lo que esencialmente consideraba un recurso comunitario y también hablando de mi empresa. Aunque aún me desa-

grada ese conflicto de intereses, la importancia de GitHub en la comunidad Git es inevitable. En lugar de un ejemplo de alojamiento Git, he decidido desarrollar esa parte del libro más detalladamente describiendo lo qué GitHub es y cómo utilizarlo de forma eficaz. Si vas a aprender a usar Git entonces saber cómo utilizar GitHub te ayudará a tomar parte en una comunidad enorme, que es valiosa no importa qué alojamiento Git decidas utilizar para tu propio código.

El otro gran cambio en el tiempo transcurrido desde la última publicación ha sido el desarrollo y aumento del protocolo HTTP para las transacciones de red de Git. La mayoría de los ejemplos en el libro han sido cambiados a HTTP desde SSH porque es mucho más sencillo.

Ha sido increíble ver a Git crecer en los últimos años a partir de un sistema de control de versiones relativamente desconocido a uno que domina básicamente el control de versiones comerciales y de código abierto. Estoy feliz de que Pro Git lo haya hecho tan bien y también haya sido capaz de ser uno de los pocos libros técnicos en el mercado que es a la vez bastante exitoso y completamente de código abierto.

Espero que disfruten de esta edición actualizada de Pro Git.

Prefacio por Ben Straub

La primera edición de este libro es lo que me enganchó a Git. Ésta fue mi introducción a un estilo de hacer software que se sentía más natural que todo lo que había visto antes. Había sido desarrollador durante varios años para entonces, pero éste fue el giro que me envió por un camino mucho más interesante que el que había seguido.

Ahora, años después, soy contribuyente a una de las principales implementaciones de Git, he trabajado para la empresa más grande de alojamiento Git, y he viajado por el mundo enseñando a la gente acerca de Git. Cuando Scott me preguntó si estaría interesado en trabajar en la segunda edición, ni siquiera me lo pensé.

Ha sido un gran placer y un privilegio trabajar en este libro. Espero que le ayude tanto como lo hizo conmigo.

Dedicatorias

A mi esposa, Becky, sin la cual esta aventura nunca hubiera comenzado. - Ben

Esta edición está dedicada a mis niñas. A mi esposa Jessica que me ha apoyado durante todos estos años y a mi hija Josefina, que me apoyará cuando esté demasiado mayor para saber lo que pasa. - Scott

Contribuidores

Debido a que este es un libro cuya traducción es “Open Source”, hemos recibido la colaboración de muchas personas a lo largo de los últimos años. A continuación hay una lista de todas las personas que han contribuido en la traducción del libro al idioma español. Muchas gracias a todos por colaborar a mejorar este libro para el beneficio de todos los hispanohablantes.

- 64 Andrés Mancera
- 36 Juan Jose Amor Iglesias
- 31 blasillo
- 15 Carlos A. Henríquez Q.
- 7 José Antonio Muñoz Jiménez
- 4 Moises Ariel Hernández Rojo
- 4 Dmunoz94
- 3 Sergio Martell
- 2 José Carlos García
- 2 Mario R. Rincón-Díaz
- 1 fatfer
- 1 Eliecer Daza
- 1 amabelster
- 1 Juan Miguel Jiménez
- 1 Juan Sebastián Casallas

Introducción

Estás a punto de pasar varias horas de tu vida leyendo acerca de Git. Dediquemos un minuto a explicar lo que tenemos preparado para tí. Éste es un breve resumen de los diez capítulos y tres apéndices de este libro.

En el **Capítulo 1**, cubriremos los Sistemas de Control de Versiones (VCSs, en sus siglas en inglés) y los fundamentos de Git -ninguna cosa técnica, sólo lo que es Git, por qué tuvo lugar en una tierra llena de VCSs, que lo diferencia, y por qué tantas personas lo están utilizando. A continuación, explicaremos cómo descargar Git y configurarlo para el primer uso si no lo tienes ya en tu sistema.

En el **Capítulo 2**, repasaremos el uso básico de Git -cómo usar Git en el 80% de los casos que encontrarás con más frecuencia. Después de leer este capítulo, deberías ser capaz de clonar un repositorio, ver lo que ha ocurrido en la historia del proyecto, modificar archivos, y contribuir cambios. Si el libro arde espontáneamente en este punto, ya deberías estar lo suficientemente ducho en el uso de Git mientras buscas otra copia.

El **Capítulo 3** trata sobre el modelo de ramificación (branching) en Git, a menudo descrito como la característica asesina de Git. Aquí aprenderás lo que realmente diferencia Git del resto. Cuando hayas terminado, puedes sentir la necesidad de pasar un momento tranquilo ponderando cómo has vivido antes de que la ramificación de Git formara parte de tu vida.

El **Capítulo 4** cubrirá Git en el servidor. Este capítulo es para aquellos que deseen configurar Git dentro de su organización o en su propio servidor personal para la colaboración. También exploraremos diversas opciones hospedadas por si prefieres dejar que otra persona lo gestione por tí.

El **Capítulo 5** repasará con todo detalle diversos flujos de trabajo distribuidos y cómo llevarlos a cabo con Git. Cuando hayas terminado con este capítulo, deberías ser capaz de trabajar como un experto con múltiples repositorios remotos, usar Git a través de correo electrónico y manejar hábilmente numerosas ramas remotas y parches aportados.

El **Capítulo 6** cubre el servicio de alojamiento GitHub e interfaz en profundidad. Cubrimos el registro y gestión de una cuenta, creación y uso de repositorios Git, flujos de trabajo comunes para contribuir a proyectos y aceptar contribu-

uciones a los tuyos, la interfaz de GitHub y un montón de pequeños consejos para hacer tu vida más fácil en general.

El **Capítulo 7** es sobre comandos avanzados de Git. Aquí aprenderás acerca de temas como el dominio del temido comando *reset*, el uso de la búsqueda binaria para identificar errores, la edición de la historia, la selección de revisión en detalle, y mucho más. Este capítulo completará tu conocimiento de Git para que puedas ser verdaderamente un maestro.

El **Capítulo 8** es sobre la configuración de tu entorno Git personalizado. Esto incluye la creación de *hook scripts* para hacer cumplir o alentar políticas personalizadas y el uso de valores de configuración de entorno para que puedas trabajar de la forma que deseas. También cubriremos la construcción de tu propio conjunto de scripts para hacer cumplir una política personalizada.

El **Capítulo 9** trata de Git y otros VCSs. Esto incluye el uso de Git en un mundo de Subversion (SVN) y la conversión de proyectos de otros VCSs a Git. Una gran cantidad de organizaciones siguen utilizando SVN y no van a cambiar, pero en este punto aprenderás el increíble poder de Git, y este capítulo te muestra cómo hacer frente si todavía tienes que utilizar un servidor SVN. También cubrimos cómo importar proyectos desde varios sistemas diferentes en caso de que convenzas a todo el mundo para dar el salto.

El **Capítulo 10** se adentra en las oscuras aunque hermosas profundidades del interior de Git. Ahora que sabes todo sobre Git y puedes manejarlo con él con poder y gracia, puedes pasar a estudiar cómo Git almacena sus objetos, qué es el modelo de objetos, detalles de *packfiles*, protocolos de servidor, y mucho más. A lo largo del libro, nos referiremos a las secciones de este capítulo por si te apetece profundizar en ese punto; pero si eres como nosotros y quieres sumergirte en los detalles técnicos, es posible que desees leer el Capítulo 10 en primer lugar. Lo dejamos a tu elección.

En el **Apéndice A** nos fijamos en una serie de ejemplos de uso de Git en diversos entornos específicos. Cubrimos un número de diferentes interfaces gráficas de usuario y entornos de programación IDE en los que es posible que desees usar Git y lo que está disponible para tí. Si estas interesado en una visión general del uso de Git en tu shell, en Visual Studio o Eclipse, echa un vistazo aquí.

En el **Apéndice B** exploramos la extensión y scripting de Git a través de herramientas como libgit2 y JGit. Si estás interesado en escribir herramientas personalizadas complejas y rápidas y necesitas acceso a bajo nivel de Git, aquí es donde puedes ver una panorámica.

Finalmente, en el **Apéndice C** repasaremos todos los comandos importantes de Git uno a uno y reseñaremos el lugar en el libro donde fueron tratados y lo que hicimos con ellos. Si quieres saber en qué parte del libro se utilizó algún comando específico de Git puedes buscarlo aquí.

Empecemos.

Table of Contents

Prefacio por Scott Chacon	iii
Prefacio por Ben Straub	v
Dedicatorias	vii
Contribuidores	ix
Introducción	xi
CHAPTER 1: Inicio - Sobre el Control de Versiones	25
Acerca del Control de Versiones	25
Sistemas de Control de Versiones Locales	26
Sistemas de Control de Versiones Centralizados	27
Sistemas de Control de Versiones Distribuidos	28
Una breve historia de Git	30
Fundamentos de Git	30
Copias instantáneas, no diferencias	31
Casi todas las operaciones son locales	32
Git tiene integridad	33
Git generalmente solo añade información	33
Los Tres Estados	33
La Línea de Comandos	35
Instalación de Git	35
Instalación en Linux	36

Instalación en Mac	36
Instalación en Windows	37
Instalación a partir del Código Fuente	38
Configurando Git por primera vez	38
Tu Identidad	39
Tu Editor	40
Comprobando tu Configuración	40
¿Cómo obtener ayuda?	41
Resumen	41
CHAPTER 2: Fundamentos de Git	43
Obteniendo un repositorio Git	43
Inicializando un repositorio en un directorio existente	43
Clonando un repositorio existente	44
Guardando cambios en el Repositorio	45
Revisando el Estado de tus Archivos	46
Rastrear Archivos Nuevos	47
Preparar Archivos Modificados	48
Estatus Abreviado	49
Ignorar Archivos	50
Ver los Cambios Preparados y No Preparados	51
Confirmar tus Cambios	54
Saltar el Área de Preparación	56
Eliminar Archivos	56
Cambiar el Nombre de los Archivos	58
Ver el Historial de Confirmaciones	59
Limitar la Salida del Historial	64
Deshacer Cosas	66
Deshacer un Archivo Preparado	67
Deshacer un Archivo Modificado	68
Trabajar con Remotos	69

Ver Tus Remotos	69
Añadir Repositorios Remotos	71
Traer y Combinar Remotos	71
Enviar a Tus Remotos	72
Inspeccionar un Remoto	73
Eliminar y Renombrar Remotos	74
Etiquetado	74
Listar Tus Etiquetas	75
Crear Etiquetas	75
Etiquetas Anotadas	76
Etiquetas Ligeras	76
Etiquetado Tardío	77
Compartir Etiquetas	78
Sacar una Etiqueta	79
Alias de Git	79
Resumen	81
CHAPTER 3: Ramificaciones en Git	83
¿Qué es una rama?	83
Crear una Rama Nueva	86
Cambiar de Rama	87
Procedimientos Básicos para Ramificar y Fusionar	91
Procedimientos Básicos de Ramificación	91
Procedimientos Básicos de Fusión	96
Principales Conflictos que Pueden Surgir en las Fusiones	98
Gestión de Ramas	101
Flujos de Trabajo Ramificados	103
Ramas de Largo Recorrido	103
Ramas Puntuales	104
Ramas Remotas	107
Publicar	112

Hacer Seguimiento a las Ramas	114
Traer y Fusionar	116
Eliminar Ramas Remotas	116
Reorganizar el Trabajo Realizado	117
Reorganización Básica	117
Algunas Reorganizaciones Interesantes	120
Los Peligros de Reorganizar	122
Reorganizar una Reorganización	125
Reorganizar vs. Fusionar	127
Recapitulación	127
CHAPTER 4: Git en el Servidor	129
Los Protocolos	130
Local Protocol	130
Protocolos HTTP	131
El Proctocolo SSH	134
El protocolo Git	135
Configurando Git en un servidor	136
Colocando un Repositorio Vacío en un Servidor	137
Pequeñas configuraciones	138
Generando tu clave pública SSH	139
Configurando el servidor	140
El demonio Git	143
HTTP Inteligente	144
GitWeb	146
GitLab	149
Instalación	149
Administración	150
Uso básico	153
Trabajando con GitLab	153
Git en un alojamiento externo	154

Resumen	155
CHAPTER 5: Git en entornos distribuidos	157
Flujos de trabajo distribuidos	157
Flujos de trabajo centralizado	157
Integration-Manager Workflow	158
Dictator and Lieutenants Workflow	159
Workflows Summary	160
Contributing to a Project	161
Commit Guidelines	162
Private Small Team	163
Private Managed Team	171
Forked Public Project	177
Public Project over E-Mail	181
Summary	184
Manteniendo un proyecto	184
Trabajando en ramas puntuales	185
Aplicando parches recibidos por e-mail	185
Recuperando ramas remotas	189
Decidiendo qué introducir	190
Integrando el trabajo de los colaboradores	192
Etiquetando tus versiones	198
Generando un número de compilación	199
Preparando una versión	200
El registro resumido	201
Resumen	201
CHAPTER 6: GitHub	203
Creación y configuración de la cuenta	203
Acceso SSH	204
Tu ícono	206

Tus direcciones de correo	207
Autentificación de dos pasos	208
Participando en Proyectos	209
Bifurcación (fork) de proyectos	209
El Flujo de Trabajo en GitHub	210
Pull Requests Avanzados	218
Markdown	223
Mantenimiento de un proyecto	229
Creación de un repositorio	229
Añadir colaboradores	231
Gestión de los Pull Requests	232
Menciones y notificaciones	237
Ficheros especiales	241
README	241
CONTRIBUTING	242
Administración del proyecto	242
Gestión de una organización	243
Conceptos básicos	244
Equipos	245
Auditorías	246
Scripting en GitHub	247
Enganches	248
La API de GitHub	252
Uso Básico	253
Comentarios en una incidencia	254
Cambio de estado de un Pull Request	255
Octokit	257
Resumen	258
CHAPTER 7: Git Tools	259
Revision Selection	259

Single Revisions	259
Short SHA-1	259
Branch References	261
RefLog Shortnames	262
Ancestry References	263
Commit Ranges	265
Interactive Staging	268
Staging and Unstaging Files	268
Staging Patches	271
Stashing and Cleaning	272
Stashing Your Work	272
Creative Stashing	275
Creating a Branch from a Stash	276
Cleaning your Working Directory	277
Signing Your Work	278
PGP Introduction	279
Signing Tags	279
Verifying Tags	280
Signing Commits	281
Everyone Must Sign	283
Searching	283
Git Grep	283
Git Log Searching	285
Rewriting History	286
Changing the Last Commit	287
Changing Multiple Commit Messages	287
Reordering Commits	290
Squashing Commits	290
Splitting a Commit	292
The Nuclear Option: filter-branch	293
Reset Demystified	295

The Three Trees	295
The Workflow	297
The Role of Reset	303
Reset With a Path	308
Squashing	311
Check It Out	314
Summary	316
Advanced Merging	317
Merge Conflicts	317
Undoing Merges	329
Other Types of Merges	332
Rerere	337
Debugging with Git	343
File Annotation	343
Binary Search	345
Submodules	347
Starting with Submodules	347
Cloning a Project with Submodules	349
Working on a Project with Submodules	351
Submodule Tips	362
Issues with Submodules	364
Bundling	366
Replace	370
Credential Storage	379
Under the Hood	380
A Custom Credential Cache	383
Summary	385
CHAPTER 8: Personalización de Git	387
Configuración de Git	387
Configuración básica del cliente	388

Colores en Git	392
Herramientas externas para fusión y diferencias	393
Formateo y espacios en blanco	396
Configuración del servidor	399
Git Attributes	400
Archivos binarios	401
Expansión de palabras clave	404
Exportación del repositorio	408
Estrategias de fusión (merge)	409
Puntos de enganche en Git	409
Instalación de un punto de enganche	410
Puntos de enganche del lado cliente	410
Puntos de enganche del lado servidor	413
Un ejemplo de implantación de una determinada política en Git	414
Punto de enganche en el lado servidor	415
Puntos de enganche del lado cliente	421
Recapitulación	425
CHAPTER 9: Git and Other Systems	427
Git as a Client	427
Git and Subversion	427
Git and Mercurial	439
Git and Perforce	448
Git and TFS	464
Migrating to Git	473
Subversion	474
Mercurial	476
Perforce	478
TFS	481
A Custom Importer	482

Summary	489
CHAPTER 10: Git Internals	491
Plumbing and Porcelain	491
Git Objects	492
Tree Objects	495
Commit Objects	498
Object Storage	501
Git References	503
The HEAD	504
Tags	505
Remotes	507
Packfiles	507
The Refspec	511
Pushing Refspecs	513
Deleting References	513
Transfer Protocols	514
The Dumb Protocol	514
The Smart Protocol	516
Protocols Summary	519
Maintenance and Data Recovery	520
Maintenance	520
Data Recovery	521
Removing Objects	524
Environment Variables	528
Global Behavior	528
Repository Locations	528
Pathspecs	529
Committing	529
Networking	530
Diffing and Merging	530

Debugging	531
Miscellaneous	533
Summary	533
Git en otros entornos	535
Embedding Git in your Applications	551
Git Commands	563
Index	581

Inicio – Sobre el Control de Versiones

1

Este capítulo habla de cómo comenzar a utilizar Git. Empezaremos describiendo algunos conceptos básicos sobre las herramientas de control de versiones; después, trataremos sobre cómo hacer que Git funcione en tu sistema; finalmente, exploraremos cómo configurarlo para empezar a trabajar con él. Al final de este capítulo deberás entender las razones por las cuales Git existe y conviene que lo uses, y deberás tener todo preparado para comenzar.

Acerca del Control de Versiones

¿Qué es un control de versiones, y por qué debería importarte? Un control de versiones es un sistema que registra los cambios realizados en un archivo o conjunto de archivos a lo largo del tiempo, de modo que puedas recuperar versiones específicas más adelante. Aunque en los ejemplos de este libro usarás archivos de código fuente como aquellos cuya versión está siendo controlada, en realidad puedes hacer lo mismo con casi cualquier tipo de archivo que encuentres en una computadora.

Si eres diseñador gráfico o de web y quieres mantener cada versión de una imagen o diseño (es algo que sin duda vas a querer), usar un sistema de control de versiones (VCS por sus siglas en inglés) es una muy buena decisión muy acertada. Dicho sistema te permite regresar a versiones anteriores de tus archivos, regresar a una versión anterior del proyecto completo, comparar cambios a lo largo del tiempo, ver quién modificó por última vez algo que pueda estar causando problemas, ver quién introdujo un problema y cuándo, y mucho más. Usar un VCS también significa generalmente que si arruinas o pierdes archivos, será posible recuperarlos fácilmente. Adicionalmente, obtendrás todos estos beneficios a un costo muy bajo.

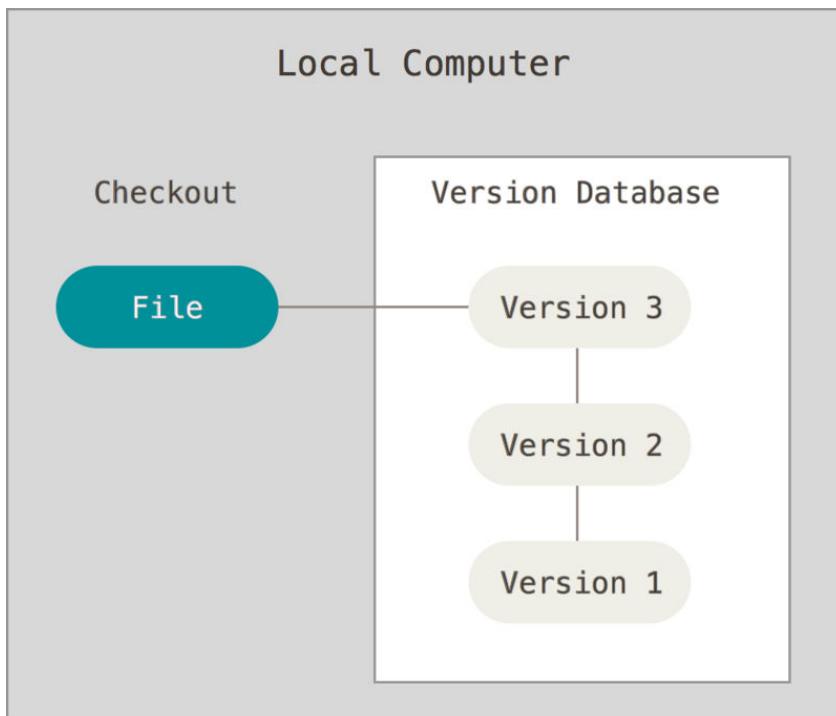
Sistemas de Control de Versiones Locales

Un método de control de versiones usado por muchas personas es copiar los archivos a otro directorio (quizás indicando la fecha y hora en que lo hicieron, si son ingeniosos). Este método es muy común porque es muy sencillo, pero también es tremadamente propenso a errores. Es fácil olvidar en qué directorio te encuentras, y guardar accidentalmente en el archivo equivocado o sobrescribir archivos que no querías.

Para afrontar este problema los programadores desarrollaron hace tiempo VCS locales que contenían una simple base de datos en la que se llevaba el registro de todos los cambios realizados a los archivos.

FIGURE 1-1

control de versiones local.



Una de las herramientas de control de versiones más popular fue un sistema llamado RCS, que todavía podemos encontrar en muchas de las computadoras actuales. Incluso el famoso sistema operativo Mac OS X incluye el comando `rcs` cuando instalas las herramientas de desarrollo. Esta herramienta funciona guardando conjuntos de parches (es decir, las diferencias entre archivos) en un

formato especial en disco, y es capaz de recrear cómo era un archivo en cualquier momento a partir de dichos parches.

Sistemas de Control de Versiones Centralizados

El siguiente gran problema con el que se encuentran las personas es que necesitan colaborar con desarrolladores en otros sistemas. Los sistemas de Control de Versiones Centralizados (CVCS por sus siglas en inglés) fueron desarrollados para solucionar este problema. Estos sistemas, como CVS, Subversion, y Perforce, tienen un único servidor que contiene todos los archivos versionados, y varios clientes que descargan los archivos desde ese lugar central. Este ha sido el estándar para el control de versiones por muchos años.

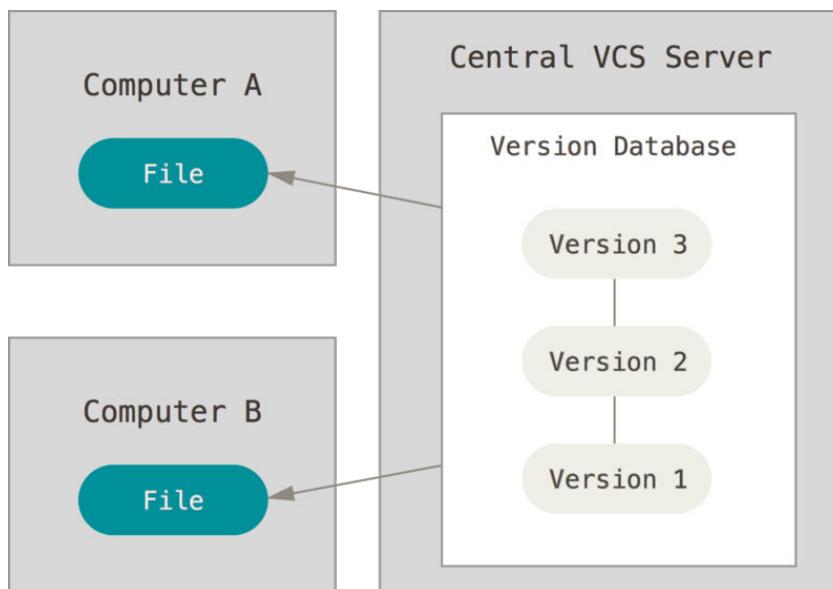


FIGURE 1-2
Control de versiones centralizado.

Esta configuración ofrece muchas ventajas, especialmente frente a VCS locales. Por ejemplo, todas las personas saben hasta cierto punto en qué están trabajando los otros colaboradores del proyecto. Los administradores tienen control detallado sobre qué puede hacer cada usuario, y es mucho más fácil administrar un CVCS que tener que lidiar con bases de datos locales en cada cliente.

Sin embargo, esta configuración también tiene serias desventajas. La más obvia es el punto único de fallo que representa el servidor centralizado. Si ese

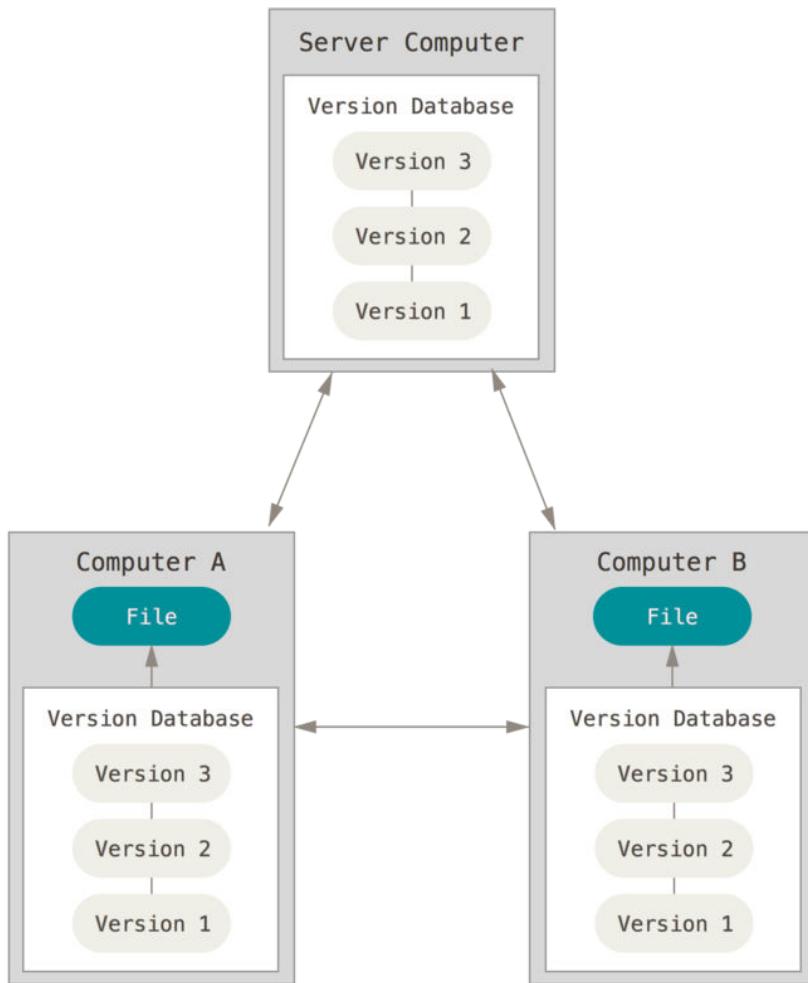
servidor se cae durante una hora, entonces durante esa hora nadie podrá colaborar o guardar cambios en archivos en los que hayan estado trabajando. Si el disco duro en el que se encuentra la base de datos central se corrompe, y no se han realizado copias de seguridad adecuadamente, se perderá toda la información del proyecto, con excepción de las copias instantáneas que las personas tengan en sus máquinas locales. Los VCS locales sufren de este mismo problema: Cuando tienes toda la historia del proyecto en un mismo lugar, te arriesgas a perderlo todo.

Sistemas de Control de Versiones Distribuidos

Los sistemas de Control de Versiones Distribuidos (DVCS por sus siglas en inglés) ofrecen soluciones para los problemas que han sido mencionados. En un DVCS (como Git, Mercurial, Bazaar o Darcs), los clientes no solo descargan la última copia instantánea de los archivos, sino que se replica completamente el repositorio. De esta manera, si un servidor deja de funcionar y estos sistemas estaban colaborando a través de él, cualquiera de los repositorios disponibles en los clientes puede ser copiado al servidor con el fin de restaurarlo. Cada clon es realmente una copia completa de todos los datos.

FIGURE 1-3

Control de versiones distribuido.



Además, muchos de estos sistemas se encargan de manejar numerosos repositorios remotos con los cuales pueden trabajar, de tal forma que puedes colaborar simultáneamente con diferentes grupos de personas en distintas maneras dentro del mismo proyecto. Esto permite establecer varios flujos de trabajo que no son posibles en sistemas centralizados, como pueden ser los modelos jerárquicos.

Una breve historia de Git

Como muchas de las grandes cosas en esta vida, Git comenzó con un poco de destrucción creativa y una gran polémica.

El kernel de Linux es un proyecto de software de código abierto con un alcance bastante amplio. Durante la mayor parte del mantenimiento del kernel de Linux (1991-2002), los cambios en el software se realizaban a través de parches y archivos. En el 2002, el proyecto del kernel de Linux empezó a usar un DVCS propietario llamado BitKeeper.

En el 2005, la relación entre la comunidad que desarrollaba el kernel de Linux y la compañía que desarrollaba BitKeeper se vino abajo, y la herramienta dejó de ser ofrecida de manera gratuita. Esto impulsó a la comunidad de desarrollo de Linux (y en particular a Linus Torvalds, el creador de Linux) a desarrollar su propia herramienta basada en algunas de las lecciones que aprendieron mientras usaban BitKeeper. Algunos de los objetivos del nuevo sistema fueron los siguientes:

- Velocidad
- Diseño sencillo
- Gran soporte para desarrollo no lineal (miles de ramas paralelas)
- Completamente distribuido
- Capaz de manejar grandes proyectos (como el kernel de Linux) eficientemente (velocidad y tamaño de los datos)

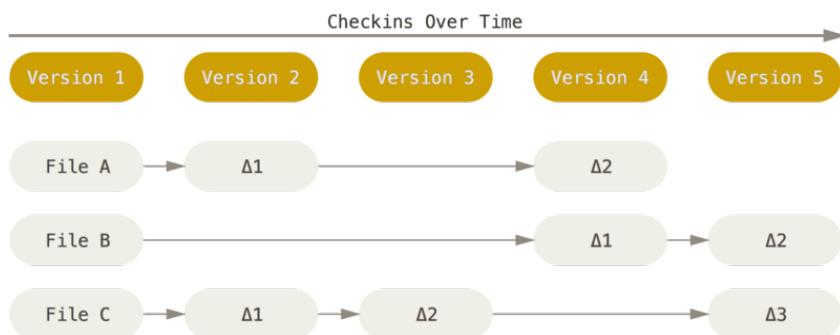
Desde su nacimiento en el 2005, Git ha evolucionado y madurado para ser fácil de usar y conservar sus características iniciales. Es tremadamente rápido, muy eficiente con grandes proyectos, y tiene un increíble sistema de ramificación (branching) para desarrollo no lineal (véase [Chapter 3](#)) (véase el [Capítulo 3](#)). [FIXME](#)

Fundamentos de Git

Entonces, ¿qué es Git en pocas palabras? Es muy importante entender bien esta sección, porque si entiendes lo que es Git y los fundamentos de cómo funciona, probablemente te será mucho más fácil usar Git efectivamente. A medida que aprendas Git, intenta olvidar todo lo que posiblemente conoces acerca de otros VCS como Subversion y Perforce. Hacer esto te ayudará a evitar confusiones sutiles a la hora de utilizar la herramienta. Git almacena y maneja la información de forma muy diferente a esos otros sistemas, a pesar de que su interfaz de usuario es bastante similar. Comprender esas diferencias evitará que te confundas a la hora de usarlo.

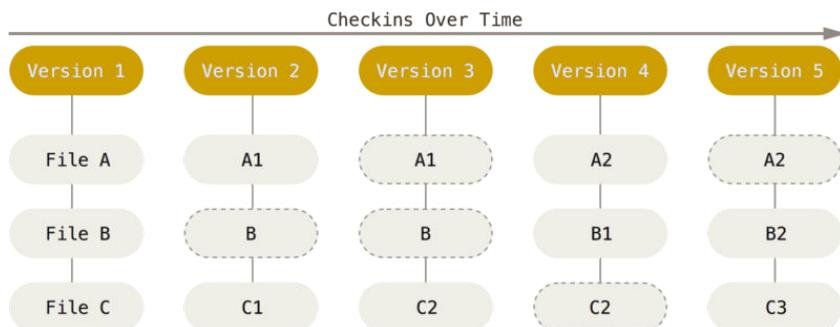
Copias instantáneas, no diferencias

La principal diferencia entre Git y cualquier otro VCS (incluyendo Subversion y sus amigos) es la forma en la que manejan sus datos. Conceptualmente, la mayoría de los otros sistemas almacenan la información como una lista de cambios en los archivos. Estos sistemas (CVS, Subversion, Perforce, Bazaar, etc.) manejan la información que almacenan como un conjunto de archivos y las modificaciones hechas a cada uno de ellos a través del tiempo.

**FIGURE 1-4**

Almacenamiento de datos como cambios en una versión de la base de cada archivo.

Git no maneja ni almacena sus datos de esta forma. Git maneja sus datos como un conjunto de copias instantáneas de un sistema de archivos miniatura. Cada vez que confirmas un cambio, o guardas el estado de tu proyecto en Git, él básicamente toma una foto del aspecto de todos tus archivos en ese momento, y guarda una referencia a esa copia instantánea. Para ser eficiente, si los archivos no se han modificado Git no almacena el archivo de nuevo, sino un enlace al archivo anterior idéntico que ya tiene almacenado. Git maneja sus datos como una secuencia de copias instantáneas.

**FIGURE 1-5**

Almacenamiento de datos como instantáneas del proyecto a través del tiempo.

Esta es una diferencia importante entre Git y prácticamente todos los demás VCS. Hace que Git reconsideré casi todos los aspectos del control de versiones que muchos de los demás sistemas copiaron de la generación anterior. Esto hace que Git se parezca más a un sistema de archivos miniatura con algunas herramientas tremadamente poderosas desarrolladas sobre él, que a un VCS. Exploraremos algunos de los beneficios que obtienes al modelar tus datos de esta manera cuando veamos ramificación (branching) en Git en el (véase [Chapter 3](#)) (véase el [Capítulo 3](#)). [FIXME](#)

Casi todas las operaciones son locales

La mayoría de las operaciones en Git sólo necesitan archivos y recursos locales para funcionar. Por lo general no se necesita información de ningún otro ordenador de tu red. Si estás acostumbrado a un CVCS donde la mayoría de las operaciones tienen el costo adicional del retardo de la red, este aspecto de Git te va a hacer pensar que los dioses de la velocidad han bendecido Git con poderes sobrenaturales. Debido a que tienes toda la historia del proyecto ahí mismo, en tu disco local, la mayoría de las operaciones parecen prácticamente inmediatas.

Por ejemplo, para navegar por la historia del proyecto, Git no necesita conectarse al servidor para obtener la historia y mostrártela - simplemente la lee directamente de tu base de datos local. Esto significa que ves la historia del proyecto casi instantáneamente. Si quieres ver los cambios introducidos en un archivo entre la versión actual y la de hace un mes, Git puede buscar el archivo hace un mes y hacer un cálculo de diferencias localmente, en lugar de tener que pedirle a un servidor remoto que lo haga u obtener una versión antigua desde la red y hacerlo de manera local.

Esto también significa que hay muy poco que no puedes hacer si estás desconectado o sin VPN. Si te subes a un avión o a un tren y quieres trabajar un poco, puedes confirmar tus cambios felizmente hasta que consigas una conexión de red para subirlos. Si te vas a casa y no consigues que tu cliente VPN funcione correctamente, puedes seguir trabajando. En muchos otros sistemas, esto es imposible o muy engorroso. En Perforce, por ejemplo, no puedes hacer mucho cuando no estás conectado al servidor. En Subversion y CVS, puedes editar archivos, pero no puedes confirmar los cambios a tu base de datos (porque tu base de datos no tiene conexión). Esto puede no parecer gran cosa, pero te sorprendería la diferencia que puede suponer.

Git tiene integridad

Todo en Git es verificado mediante una suma de comprobación (checksum en inglés) antes de ser almacenado, y es identificado a partir de ese momento mediante dicha suma. Esto significa que es imposible cambiar los contenidos de cualquier archivo o directorio sin que Git lo sepa. Esta funcionalidad está integrada en Git al más bajo nivel y es parte integral de su filosofía. No puedes perder información durante su transmisión o sufrir corrupción de archivos sin que Git sea capaz de detectarlo.

El mecanismo que usa Git para generar esta suma de comprobación se conoce como hash SHA-1. Se trata de una cadena de 40 caracteres hexadecimales (0-9 y a-f), y se calcula en base a los contenidos del archivo o estructura del directorio en Git. Un hash SHA-1 se ve de la siguiente forma:

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

Verás estos valores hash por todos lados en Git porque son usados con mucha frecuencia. De hecho, Git guarda todo no por nombre de archivo, sino por el valor hash de sus contenidos.

Git generalmente solo añade información

Cuando realizas acciones en Git, casi todas ellas solo añaden información a la base de datos de Git. Es muy difícil conseguir que el sistema haga algo que no se pueda enmendar, o que de algún modo borre información. Como en cualquier VCS, puedes perder o estropear cambios que no has confirmado todavía. Pero después de confirmar una copia instantánea en Git es muy difícil de perderla, especialmente si envías tu base de datos a otro repositorio con regularidad.

Esto hace que usar Git sea un placer, porque sabemos que podemos experimentar sin peligro de estropear gravemente las cosas. Para un análisis más exhaustivo de cómo almacena Git su información y cómo puedes recuperar datos aparentemente perdidos, ver “[Deshacer Cosas](#)” Capítulo 2. [FIXME](#)

Los Tres Estados

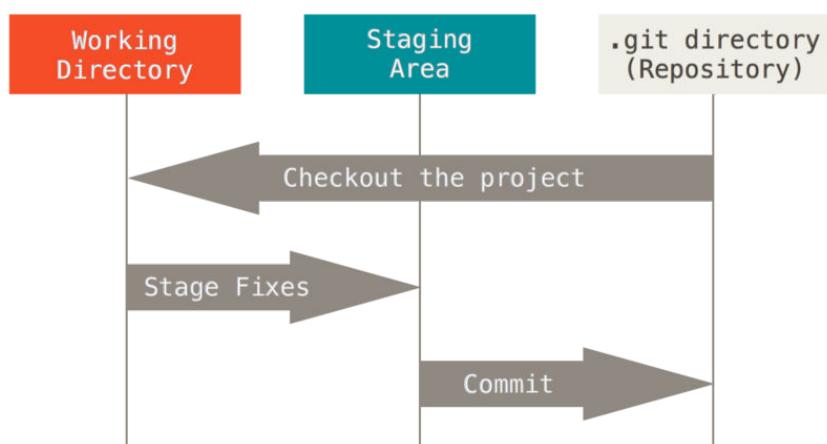
Ahora presta atención. Esto es lo más importante que debes recordar acerca de Git si quieras que el resto de tu proceso de aprendizaje prosiga sin problemas. Git tiene tres estados principales en los que se pueden encontrar tus archivos: confirmado (committed), modificado (modified), y preparado (staged). Confirmado significa que los datos están almacenados de manera segura en tu base de datos local. Modificado significa que has modificado el archivo pero todavía

no lo has confirmado a tu base de datos. Preparado significa que has marcado un archivo modificado en su versión actual para que vaya en tu próxima confirmación.

Esto nos lleva a las tres secciones principales de un proyecto de Git: El directorio de Git (Git directory), el directorio de trabajo (working directory), y el área de preparación (staging area).

FIGURE 1-6

Directorio de trabajo, área de almacenamiento, y el directorio Git.



El directorio de Git es donde se almacenan los metadatos y la base de datos de objetos para tu proyecto. Es la parte más importante de Git, y es lo que se copia cuando clonas un repositorio desde otra computadora.

El directorio de trabajo es una copia de una versión del proyecto. Estos archivos se sacan de la base de datos comprimida en el directorio de Git, y se colocan en disco para que los puedas usar o modificar.

El área de preparación es un archivo, generalmente contenido en tu directorio de Git, que almacena información acerca de lo que va a ir en tu próxima confirmación. A veces se le denomina índice (“index”), pero se está convirtiendo en estándar el referirse a ella como el área de preparación.

El flujo de trabajo básico en Git es algo así:

1. Modificas una serie de archivos en tu directorio de trabajo.
2. Preparas los archivos, añadiéndolos a tu área de preparación.
3. Confirmas los cambios, lo que toma los archivos tal y como están en el área de preparación y almacena esa copia instantánea de manera permanente en tu directorio de Git.

Si una versión concreta de un archivo está en el directorio de Git, se considera confirmada (committed). Si ha sufrido cambios desde que se obtuvo del repositorio, pero ha sido añadida al área de preparación, está preparada (staged). Y si ha sufrido cambios desde que se obtuvo del repositorio, pero no se ha preparado, está modificada (modified). En el **Chapter 2 Capítulo 2** aprenderás más acerca de estos estados y de cómo puedes aprovecharlos o saltarte toda la parte de preparación.

La Línea de Comandos

Existen muchas formas de usar Git. Por un lado tenemos las herramientas originales de línea de comandos, y por otro lado tenemos una gran variedad de interfaces de usuario con distintas capacidades. En ese libro vamos a utilizar Git desde la línea de comandos. La línea de comandos es el único lugar en donde puedes ejecutar **todos** los comandos de Git - la mayoría de interfaces gráficas de usuario solo implementan una parte de las características de Git por motivos de simplicidad. Si tú sabes cómo realizar algo desde la línea de comandos, seguramente serás capaz de averiguar cómo hacer lo mismo desde una interfaz gráfica. Sin embargo, la relación opuesta no es necesariamente cierta. Así mismo, la decisión de qué cliente gráfico utilizar depende totalmente de tu gusto, pero *todos* los usuarios tendrán las herramientas de línea de comandos instaladas y disponibles.

Nosotros esperamos que sepas cómo abrir el Terminal en Mac, o el “Command Prompt” o “Powershell” en Windows. Si no entiendes de lo que estamos hablando aquí, te recomendamos que hagas una pausa para investigar acerca de esto de tal forma que puedas entender el resto de las explicaciones y descripciones que siguen en este libro.

Instalación de Git

Antes de empezar a utilizar Git, tienes que instalarlo en tu computadora. Incluso si ya está instalado, este es posiblemente un buen momento para actualizarlo a su última versión. Puedes instalarlo como un paquete, a partir de un archivo instalador, o bajando el código fuente y compilándolo tú mismo.

EXAMPLE 1-1.

Este libro fue escrito utilizando la versión **2.0.0** de Git. Aun cuando la mayoría de comandos que usaremos deben funcionar en versiones más antiguas de Git, es posible que algunos de ellos no funcionen o funcionen ligeramente diferente si estás utilizando una versión anterior de Git. Debido a que Git es particu-

larmente bueno en preservar compatibilidad hacia atrás, cualquier versión posterior a 2.0 debe funcionar bien.

Instalación en Linux

Si quieres instalar Git en Linux a través de un instalador binario, en general puedes hacerlo mediante la herramienta básica de administración de paquetes que trae tu distribución. Si estás en Fedora por ejemplo, puedes usar yum:

```
$ yum install git
```

Si estás en una distribución basada en Debian como Ubuntu, puedes usar apt-get:

```
$ apt-get install git
```

Para opciones adicionales, la página web de Git tiene instrucciones para la instalación en diferentes tipos de Unix. Puedes encontrar esta información en <http://git-scm.com/download/linux>.

Instalación en Mac

Hay varias maneras de instalar Git en un Mac. Probablemente la más sencilla es instalando las herramientas Xcode de Línea de Comandos. En Mavericks (10.9) o superior puedes hacer esto desde el Terminal si intentas ejecutar *git* por primera vez. Si no lo tienes instalado, te preguntará si deseas instalarlo.

Si deseas una versión más actualizada, puedes hacerlo partir de un instalador binario. Un instalador de Git para OSX es mantenido en la página web de Git. Lo puedes descargar en <http://git-scm.com/download/mac>.

**FIGURE 1-7**

Instalador de Git en OS X.

También puedes instalarlo como parte del instalador de Github para Mac. Su interfaz gráfica de usuario tiene la opción de instalar las herramientas de línea de comandos. Puedes descargar esa herramienta desde el sitio web de Github para Mac en <http://mac.github.com>.

Instalación en Windows

También hay varias maneras de instalar Git en Windows. La forma más oficial está disponible para ser descargada en el sitio web de Git. Solo tienes que visitar <http://git-scm.com/download/win> y la descarga empezará automáticamente. Fíjate que éste es un proyecto conocido como Git para Windows (también llamado msysGit), el cual es diferente de Git. Para más información acerca de este proyecto visita <http://msysgit.github.io/>.

Otra forma de obtener Git fácilmente es mediante la instalación de GitHub para Windows. El instalador incluye la versión de línea de comandos y la interfaz de usuario de Git. Además funciona bien con Powershell y establece correctamente “caching” de credenciales y configuración CRLF adecuada. Aprendaremos acerca de todas estas cosas un poco más adelante, pero por ahora es suficiente mencionar que éstas son cosas que deseas. Puedes descargar este instalador del sitio web de GitHub para Windows en <http://windows.github.com>.

Instalación a partir del Código Fuente

Algunas personas desean instalar Git a partir de su código fuente debido a que obtendrás una versión más reciente. Los instaladores binarios tienen a estar un poco atrasados. Sin embargo, esto ha hecho muy poca diferencia a medida que Git ha madurado en los últimos años.

Para instalar Git desde el código fuente necesitas tener las siguientes librerías de las que Git depende: curl, zlib, openssl, expat y libiconv. Por ejemplo, si estás en un sistema que tiene yum (como Fedora) o apt-get (como un sistema basado en Debian), puedes usar estos comandos para instalar todas las dependencias:

```
$ yum install curl-devel expat-devel gettext-devel \
openssl-devel zlib-devel
```

```
$ apt-get install libcurl4-gnutls-dev libexpat1-dev gettext \
libbz-dev libssl-dev
```

Cuando tengas todas las dependencias necesarias, puedes descargar la versión más reciente de Git en diferentes sitios. Puedes obtenerlo a partir del sitio Kernel.org en <https://www.kernel.org/pub/software/scm/git>, o su “mirror” en el sitio web de GitHub en <https://github.com/git/git/releases>. Generalmente la más reciente versión en la página web de GitHub es un poco mejor, pero la página de kernel.org también tiene ediciones con firma en caso de que deseas verificar tu descarga.

Luego tienes que compilar e instalar de la siguiente manera:

```
$ tar -zxf git-2.0.0.tar.gz
$ cd git-2.0.0
$ make config
$ ./config --prefix=/usr
$ make all doc info
$ sudo make install install-doc install-html install-info
```

Una vez hecho esto, también puedes obtener Git, a través del propio Git, para futuras actualizaciones:

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

Configurando Git por primera vez

Ahora que tienes Git en tu sistema, vas a querer hacer algunas cosas para personalizar tu entorno de Git. Es necesario hacer estas cosas solamente una vez en tu computadora, y se mantendrán entre actualizaciones. También puedes

cambiarlas en cualquier momento volviendo a ejecutar los comandos correspondientes.

Git trae una herramienta llamada `git config` que te permite obtener y establecer variables de configuración que controlan el aspecto y funcionamiento de Git. Estas variables pueden almacenarse en tres sitios distintos:

1. Archivo `/etc/gitconfig`: Contiene valores para todos los usuarios del sistema y todos sus repositorios. Si pasas la opción `--system` a `git config`, lee y escribe específicamente en este archivo.
2. Archivo `~/.gitconfig` o `~/.config/git/config`: Este archivo es específico a tu usuario. Puedes hacer que Git lea y escriba específicamente en este archivo pasando la opción `--global`.
3. Archivo `config` en el directorio de Git (es decir, `.git/config`) del repositorio que estés utilizando actualmente: Este archivo es específico al repositorio actual.

Cada nivel sobrescribe los valores del nivel anterior, por lo que los valores de `.git/config` tienen preferencia sobre los de `/etc/gitconfig`.

En sistemas Windows, Git busca el archivo `.gitconfig` en el directorio `$HOME` (para mucha gente será `(C:\Users\$USER)`). También busca el archivo `/etc/gitconfig`, aunque esta ruta es relativa a la raíz MSys, que es donde decidiste instalar Git en tu sistema Windows cuando ejecutaste el instalador.

Tu Identidad

Lo primero que deberás hacer cuando instalas Git es establecer tu nombre de usuario y dirección de correo electrónico. Esto es importante porque los “commits” de Git usan esta información, y es introducida de manera inmutable en los commits que envías:

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

De nuevo, solo necesitas hacer esto una vez si especificas la opción `--global`, ya que Git siempre usará esta información para todo lo que hagas en ese sistema. Si quieres sobrescribir esta información con otro nombre o dirección de correo para proyectos específicos, puedes ejecutar el comando sin la opción `--global` cuando estés en ese proyecto.

Muchas de las herramientas de interfaz gráfica te ayudarán a hacer esto la primera vez que las uses.

Tu Editor

Ahora que tu identidad está configurada, puedes elegir el editor de texto por defecto que se utilizará cuando Git necesite que introduzcas un mensaje. Si no indicas nada, Git usa el editor por defecto de tu sistema, que generalmente es Vim. Si quieres usar otro editor de texto como Emacs, puedes hacer lo siguiente:

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

Vim y Emacs son editores de texto frecuentemente usados por desarrolladores en sistemas basados en Unix como Linux y Mac. Si no estás familiarizado con ninguno de estos editores o estás en un sistema Windows, es posible que necesites buscar instrucciones acerca de cómo configurar tu editor favorito con Git. Si no configuras un editor así y no conoces acerca de Vim o Emacs, es muy factible que termines en un estado bastante confuso en el momento en que sean ejecutados.

Comprobando tu Configuración

Si quieres comprobar tu configuración, puedes usar el comando `git config --list` para mostrar todas las propiedades que Git ha configurado:

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

Puede que veas claves repetidas, porque Git lee la misma clave de distintos archivos (`/etc/gitconfig` y `~/.gitconfig`, por ejemplo). En ese caso, Git usa el último valor para cada clave única que ve.

También puedes comprobar qué valor que Git utilizará para una clave específica ejecutando `git config <key>`:

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

¿Cómo obtener ayuda?

Si alguna vez necesitas ayuda usando Git, existen tres formas de ver la página del manual (manpage) para cualquier comando de Git:

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

Por ejemplo, puedes ver la página del manual para el comando config ejecutando

```
$ git help config
```

Estos comandos son muy útiles porque puedes acceder a ellos desde cualquier sitio, incluso sin conexión. Si las páginas del manual y este libro no son suficientes y necesitas que te ayude una persona, puedes probar en los canales #git o #github del servidor de IRC Freenode (irc.freenode.net). Estos canales están llenos de cientos de personas que conocen muy bien Git y suelen estar dispuestos a ayudar.

Resumen

Para este momento debes tener una comprensión básica de lo que es Git, y de cómo se diferencia de cualquier otro sistema de control de versiones centralizado que pudieras haber utilizado previamente. De igual manera, Git debe estar funcionando en tu sistema y configurado con tu identidad personal. Es hora de aprender los fundamentos de Git.

Fundamentos de Git

2

Si pudieras leer solo un capítulo para empezar a trabajar con Git, este es el capítulo que debes leer. Este capítulo cubre todos los comandos básicos que necesitas para hacer la gran mayoría de cosas a las que eventualmente vas a dedicar tu tiempo mientras trabajas con Git. Al final del capítulo, deberás ser capaz de configurar e inicializar un repositorio, comenzar y detener el seguimiento de archivos, y preparar (stage) y confirmar (commit) cambios. También te enseñaremos a configurar Git para que ignore ciertos archivos y patrones, cómo enmendar errores rápida y fácilmente, cómo navegar por la historia de tu proyecto y ver cambios entre confirmaciones, y cómo enviar (push) y recibir (pull) de repositorios remotos.

Obteniendo un repositorio Git

Puedes obtener un proyecto Git de dos maneras. La primera es tomar un proyecto o directorio existente e importarlo en Git. La segunda es clonar un repositorio existente en Git desde otro servidor.

Inicializando un repositorio en un directorio existente

Si estás empezando a seguir un proyecto existente en Git, debes ir al directorio del proyecto y usar el siguiente comando:

```
$ git init
```

Esto crea un subdirectorio nuevo llamado `.git`, el cual contiene todos los archivos necesarios del repositorio – un esqueleto de un repositorio de Git. Todavía no hay nada en tu proyecto que esté bajo seguimiento. Puedes revisar [Chapter 10](#) para obtener más información acerca de los archivos presentes en el directorio `.git` que acaba de ser creado.

Si deseas empezar a controlar versiones de archivos existentes (a diferencia de un directorio vacío), probablemente deberías comenzar el seguimiento de esos archivos y hacer una confirmación inicial. Puedes conseguirlo con unos pocos comandos `git add` para especificar qué archivos quieres controlar, seguidos de un `git commit` para confirmar los cambios:

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

Veremos lo que hacen estos comandos más adelante. En este momento, tienes un repositorio de Git con archivos bajo seguimiento y una confirmación inicial.

Clonando un repositorio existente

Si deseas obtener una copia de un repositorio Git existente — por ejemplo, un proyecto en el que te gustaría contribuir — el comando que necesitas es `git clone`. Si estás familiarizado con otros sistemas de control de versiones como Subversion, verás que el comando es “clone” en vez de “checkout”. Es una distinción importante, ya que Git recibe una copia de casi todos los datos que tiene el servidor. Cada versión de cada archivo de la historia del proyecto es descargada por defecto cuando ejecutas `git clone`. De hecho, si el disco de tu servidor se corrompe, puedes usar cualquiera de los clones en cualquiera de los clientes para devolver al servidor al estado en el que estaba cuando fue clonado (puede que pierdas algunos hooks del lado del servidor y demás, pero toda la información acerca de las versiones estará ahí) — véase “[Configurando Git en un servidor](#)” para más detalles.

Puedes clonar un repositorio con `git clone [url]`. Por ejemplo, si quieres clonar la librería de Git llamada libgit2 puedes hacer algo así:

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

Esto crea un directorio llamado `libgit2`, inicializa un directorio `.git` en su interior, descarga toda la información de ese repositorio y saca una copia de trabajo de la última versión. Si te metes en el directorio `libgit2`, verás que están los archivos del proyecto listos para ser utilizados. Si quieres clonar el repositorio a un directorio con otro nombre que no sea `libgit2`, puedes especificarlo con la siguiente opción de línea de comandos:

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

Ese comando hace lo mismo que el anterior, pero el directorio de destino se llamará `mylibgit`.

Git te permite usar distintos protocolos de transferencia. El ejemplo anterior usa el protocolo `https://`, pero también puedes utilizar `git://` o `usuario@servidor:ruta/del/repositorio.git` que utiliza el protocolo de transferencia SSH. En “[Configurando Git en un servidor](#)” se explicarán todas las opciones disponibles a la hora de configurar el acceso a tu repositorio de Git, y las ventajas e inconvenientes de cada una.

Guardando cambios en el Repositorio

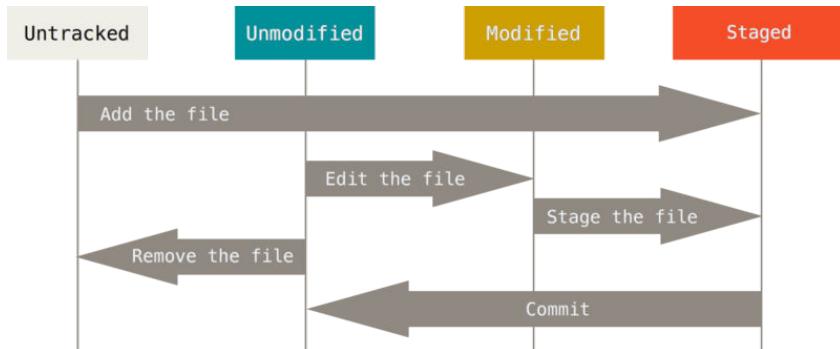
Ya tienes un repositorio Git y un *checkout* o copia de trabajo de los archivos de dicho proyecto. El siguiente paso es realizar algunos cambios y confirmar instantáneas de esos cambios en el repositorio cada vez que el proyecto alcance un estado que quieras conservar.

Recuerda que cada archivo de tu repositorio puede tener dos estados: rastreados y sin rastrear. Los archivos rastreados (*tracked files* en inglés) son todos aquellos archivos que estaban en la última instantánea del proyecto; pueden ser archivos sin modificar, modificados o preparados. Los archivos sin rastrear son todos los demás - cualquier otro archivo en tu directorio de trabajo que no estaba en tu última instantánea y que no están en el área de preparación (*staging area*). Cuando clonas por primera vez un repositorio, todos tus archivos estarán rastreados y sin modificar pues acabas de sacarlos y aun no han sido editados.

Mientras editas archivos, Git los ve como modificados, pues han sido cambiados desde su último *commit*. Luego preparas estos archivos modificados y finalmente confirmas todos los cambios preparados, y repites el ciclo.

FIGURE 2-1

El ciclo de vida del estado de tus archivos.



Revisando el Estado de tus Archivos

La herramienta principal para determinar qué archivos están en qué estado es el comando `git status`. Si ejecutas este comando inmediatamente después de clonar un repositorio, deberías ver algo como esto:

```
$ git status
On branch master
nothing to commit, working directory clean
```

Esto significa que tienes un directorio de trabajo limpio - en otras palabras, que no hay archivos rastreados y modificados. Además, Git no encuentra ningún archivo sin rastrear, de lo contrario aparecerían listados aquí. Finalmente, el comando te indica en cuál rama estás y te informa que no ha variado con respecto a la misma rama en el servidor. Por ahora, la rama siempre será “master”, que es la rama por defecto; no le prestaremos atención ahora. **Chapter 3** tratará en detalle las ramas y las referencias.

Supongamos que añades un nuevo archivo a tu proyecto, un simple README. Si el archivo no existía antes, y ejecutas `git status`, verás el archivo sin rastrear de la siguiente manera:

```
$ echo 'My Project' > README
$ git status
On branch 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)
```

Puedes ver que el archivo README está sin rastrear porque aparece debajo del encabezado “Untracked files” (“Archivos no rastreados” en inglés) en la salida. Sin rastrear significa que Git ve archivos que no tenías en el *commit* anterior. Git no los incluirá en tu próximo *commit* a menos que se lo indiques explícitamente. Se comporta así para evitar incluir accidentalmente archivos binarios o cualquier otro archivo que no quieras incluir. Como tú sí quieres incluir README, debes comenzar a rastrearlo.

Rastrear Archivos Nuevos

Para comenzar a rastrear un archivo debes usar el comando `git add`. Para comenzar a rastrear el archivo README, puedes ejecutar lo siguiente:

```
$ git add README
```

Ahora si vuelves a ver el estado del proyecto, verás que el archivo README está siendo rastreado y está preparado para ser confirmado:

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

    new file:   README
```

Puedes ver que está siendo rastreado porque aparece luego del encabezado “Cambios a ser confirmados” (“Changes to be committed” en inglés). Si confirmas en este punto, se guardará en el historial la versión del archivo correspondiente al instante en que ejecutaste `git add`. Anteriormente cuando ejecutaste `git init`, ejecutaste luego `git add (files)` - lo cual inició el rastreo de archivos en tu directorio. El comando `git add` puede recibir tanto una ruta de archivo como de un directorio; si es de un directorio, el comando añade recursivamente los archivos que están dentro de él.

Preparar Archivos Modificados

Vamos a cambiar un archivo que esté rastreado. Si cambias el archivo rastreado llamado “CONTRIBUTING.md” y luego ejecutas el comando `git status`, verás algo parecido a esto:

```
$ git status
On branch 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
```

El archivo “CONTRIBUTING.md” aparece en una sección llamada “Changes not staged for commit” (“Cambios no preparado para confirmar” en inglés) - lo que significa que existe un archivo rastreado que ha sido modificado en el directorio de trabajo pero que aun no está preparado. Para prepararlo, ejecutas el comando `git add`. `git add` es un comando que cumple varios propósitos - lo usas para empezar a rastrear archivos nuevos, preparar archivos, y hacer otras cosas como marcar como resuelto archivos en conflicto por combinación. Es más útil que lo veas como un comando para “añadir este contenido a la próxima confirmación” mas que para “añadir este archivo al proyecto”. Ejecutemos `git add` para preparar el archivo “CONTRIBUTING.md” y luego ejecutemos `git status`:

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

    new file: README
    modified: CONTRIBUTING.md
```

Ambos archivos están preparados y formarán parte de tu próxima confirmación. En este momento, supongamos que recuerdas que debes hacer un pequeño cambio en `CONTRIBUTING.md` antes de confirmarlo. Abres de nuevo el archi-

vo, lo cambias y ahora estás listos para confirmar. Sin embargo, ejecutemos `git status` una vez más:

```
$ vim CONTRIBUTING.md
$ git status
On branch 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
```

¡¿Pero qué...?! Ahora `CONTRIBUTING.md` aparece como preparado y como no preparado. ¿Cómo es posible? Resulta que Git prepara un archivo de acuerdo al estado que tenía cuando ejecutas el comando `git add`. Si confirmas ahora, se confirmará la versión de `CONTRIBUTING.md` que tenías la última vez que ejecutaste `git add` y no la versión que ves ahora en tu directorio de trabajo al ejecutar `git commit`. Si modificas un archivo luego de ejecutar `git add`, deberás ejecutar `git add` de nuevo para preparar la última versión del archivo:

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

    new file:   README
    modified:   CONTRIBUTING.md
```

Estatus Abreviado

Si bien es cierto que la salida de `git status` es bastante explícita, también es verdad que es muy extensa. Git ofrece una opción para obtener un estatus abreviado, de manera que puedas ver tus cambios de una forma más compacta. Si ejecutas `git status -s` o `git status --short` obtendrás una salida mucho más simplificada.

```
$ git status -s
M README
MM Rakefile
A lib/git.rb
M lib/simplegit.rb
?? LICENSE.txt
```

Los archivos nuevos que no están rastreados tienen un ?? a su lado, los archivos que están preparados tienen una A y los modificados una M. El estado aparece en dos columnas - la columna de la izquierda indica el estado preparado y la columna de la derecha indica el estado sin preparar. Por ejemplo, en esa salida, el archivo README está modificado en el directorio de trabajo pero no está preparado, mientras que lib/simplegit.rb está modificado y preparado. El archivo Rakefile fue modificado, preparado y modificado otra vez por lo que existen cambios preparados y sin preparar.

Ignorar Archivos

A veces, tendrás algún tipo de archivo que no quieras que Git añada automáticamente o más aun, que ni siquiera quieras que aparezca como no rastreado. Este suele ser el caso de archivos generados automáticamente como trazas o archivos creados por tu sistema de construcción. En estos casos, puedes crear un archivo llamado `.gitignore` que liste patrones a considerar. Este es un ejemplo de un archivo `.gitignore`:

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

La primera línea le indica a Git que ignore cualquier archivo que termine en “.o” o “.a” - archivos de objeto o librerías que pueden ser producto de compilar tu código. La segunda línea le indica a Git que ignore todos los archivos que terminen con una tilde (~), lo cual es usado por varios editores de texto como Emacs para marcar archivos temporales. También puedes incluir cosas como trazas, temporales, o pid directamente; documentación generada automáticamente; etc. Crear un archivo `.gitignore` antes de comenzar a trabajar es generalmente una buena idea pues así evitas confirmar accidentalmente archivos que en realidad no quieras incluir en tu repositorio Git.

Las reglas sobre los patrones que puedes incluir en el archivo `.gitignore` son las siguientes:

- Ignorar las líneas en blanco y aquellas que comiencen con #.
- Aceptar patrones glob estándar.
- Los patrones pueden terminar en barra (/) para especificar un directorio.
- Los patrones pueden negarse si se añade al principio el signo de exclamación (!).

Los patrones glob son una especie de expresión regular simplificada usada por los terminales. Un asterisco (*) corresponde a cero o más caracteres; [abc] corresponde a cualquier carácter dentro de los corchetes (en este caso a, b o c); el signo de interrogación (?) corresponde a un carácter cualquier; y los corchetes sobre caracteres separados por un guión ([0-9]) corresponde a cualquier carácter entre ellos (en este caso del 0 al 9). También puedes usar dos asteriscos para indicar directorios anidados; a/**/z coincide con a/z, a/b/z, a/b/c/z, etc.

Aquí puedes ver otro ejemplo de un archivo .gitignore:

```
# ignora los archivos terminados en .a
*.a

# pero no lib.a, aun cuando habia ignorado los archivos terminados en .a en la linea anterior
!lib.a

# ignora únicamente el archivo TODO de la raiz, no subdir/TODO
/TODO

# ignora todos los archivos del directorio build/
build/

# ignora doc/notes.txt, pero este no doc/server/arch.txt
doc/*.txt

# ignora todos los archivos .txt el directorio doc/
doc/**/*.txt
```

GitHub mantiene una extensa lista de archivos .gitignore adecuados a docenas de proyectos y lenguajes en <https://github.com/github/gitignore> en caso de que quieras tener un punto de partida para tu proyecto.

Ver los Cambios Preparados y No Preparados

Si el comando `git status` es muy impreciso para ti - quieres ver exactamente que ha cambiado, no solo cuáles archivos lo han hecho - puedes usar el comando `git diff`. Hablaremos sobre `git diff` más adelante, pero lo usarás pro-

bablemente para responder estas dos preguntas: ¿Qué has cambiado pero aun no has preparado? y ¿Qué has preparado y está listo para confirmar? A pesar de que `git status` responde a estas preguntas de forma muy general listando el nombre de los archivos, `git diff` te muestra las líneas exactas que fueron añadidas y eliminadas, es decir, el parche.

Supongamos que editas y preparas el archivo README de nuevo y luego editas CONTRIBUTING.md pero no lo preparas. Si ejecutas el comando `git status`, verás algo como esto:

```
$ git status
On branch 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
```

Para ver qué has cambiado pero aun no has preparado, escribe `git diff` sin más parámetros:

```
$ 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
```

Este comando compara lo que tienes en tu directorio de trabajo con lo que está en el área de preparación. El resultado te indica los cambios que has hecho pero que aun no has preparado.

Siquieres ver lo que has preparado y será incluido en la próxima confirmación, puedes usar `git diff --staged`. Este comando compara tus cambios preparados con la última instantánea confirmada.

```
$ git diff --staged
diff --git a/README b/README
new file mode 100644
index 000000..03902a1
--- /dev/null
+++ b/README
@@ -0,0 +1 @@
+My Project
```

Es importante resaltar que al llamar a `git diff` sin parámetros no verás los cambios desde tu última confirmación - solo verás los cambios que aun no están preparados. Esto puede ser confuso porque si preparas todos tus cambios, `git diff` no te devolverá ninguna salida.

Pasemos a otro ejemplo, si preparas el archivo `CONTRIBUTING.md` y luego lo editas, puedes usar `git diff` para ver los cambios en el archivo que están preparados y los cambios que no lo están. Si nuestro ambiente es como este:

```
$ git add CONTRIBUTING.md
$ echo 'test line' >> CONTRIBUTING.md
$ git status
On branch 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
```

Puedes usar `git diff` para ver qué está sin preparar

```
$ 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/PROJE
+#+ test line
```

y `git diff --cached` para ver que has preparado hasta ahora (`--staged` y `--cached` son sinónimos):

```
$ 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 COMO HERRAMIENTA EXTERNA

A lo largo del libro, continuaremos usando el comando `git diff` de distintas maneras. Existe otra forma de ver estas diferencias si prefieres utilizar una interfaz gráfica u otro programa externo. Si ejecutas `git difftool` en vez de `git diff`, podrás ver los cambios con programas de este tipo como Araxis, emerge, vimdiff y más. Ejecuta `git difftool --tool-help` para ver qué tienes disponible en tu sistema.

Confirmar tus Cambios

Ahora que tu área de preparación está como quieras, puedes confirmar tus cambios. Recuerda que cualquier cosa que no esté preparada - cualquier archivo que hayas creado o modificado y que no hayas agregado con `git add` desde su edición - no será confirmado. Se mantendrán como archivos modificados en tu disco. En este caso, digamos que la última vez que ejecutaste `git status` verificaste que todo estaba preparado y que estás listos para confirmar tus cambios. La forma más sencilla de confirmar es escribiendo `git commit`:

```
$ git commit
```

Al hacerlo, arrancará el editor de tu preferencia. (El editor se establece a través de la variable de ambiente `$EDITOR` de tu terminal - usualmente es vim o emacs, aunque puedes configurarlo con el editor que quieras usando el comando `git config --global core.editor` tal como viste en [Chapter 1](#)).

El editor mostrará el siguiente texto (este ejemplo corresponde a una pantalla de Vim):

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#   new file: README
#   modified: CONTRIBUTING.md
#
#
#
#
".git/COMMIT_EDITMSG" 9L, 283C
```

Puedes ver que el mensaje de confirmación por defecto contiene la última salida del comando `git status` comentada y una línea vacía encima de ella. Puedes eliminar estos comentarios y escribir tu mensaje de confirmación, o puedes dejarlos allí para ayudarte a recordar qué estás confirmando. (Para obtener una forma más explícita de recordar qué has modificado, puedes pasar la opción `-v` a `git commit`. Al hacerlo se incluirá en el editor el diff de tus cambios para que veas exactamente qué cambios estás confirmando.) Cuando sales del editor, Git crea tu confirmación con tu mensaje (eliminando el texto comentado y el diff).

Otra alternativa es escribir el mensaje de confirmación directamente en el comando `commit` utilizando la opción `-m`:

```
$ 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
```

¡Has creado tu primera confirmación (o *commit*)! Puedes ver que la confirmación te devuelve una salida descriptiva: indica cuál rama has confirmado (`master`), que *checksum* SHA-1 tiene el *commit* (`463dc4f`), cuántos archivos

han cambiado y estadísticas sobre las líneas añadidas y eliminadas en el *commit*.

Recuerda que la confirmación guarda una instantánea de tu área de preparación. Todo lo que no hayas preparado sigue allí modificado; puedes hacer una nueva confirmación para añadirlo a tu historial. Cada vez que realizas un *commit*, guardas una instantánea de tu proyecto la cual puedes usar para comparar o volver a ella luego.

Saltar el Área de Preparación

A pesar de que puede resultar muy útil para ajustar los *commits* tal como quieres, el área de preparación es a veces un paso más complejo a lo que necesitas para tu flujo de trabajo. Si quieres saltarte el área de preparación, Git te ofrece un atajo sencillo. Añadiendo la opción `-a` al comando `git commit` harás que Git prepare automáticamente todos los archivos rastreados antes de confirmarlos, ahorrándote el paso de `git add`:

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

    modified:   CONTRIBUTING.md

no changes added to commit (use "git add" and/or "git commit -a")
$ git commit -a -m 'added new benchmarks'
[master 83e38c7] added new benchmarks
 1 file changed, 5 insertions(+), 0 deletions(-)
```

Fíjate que en este caso no fue necesario ejecutar `git add` sobre el archivo `CONTRIBUTING.md` antes de confirmar.

Eliminar Archivos

Para eliminar archivos de Git, debes eliminarlos de tus archivos rastreados (o mejor dicho, eliminarlos del área de preparación) y luego confirmar. Para ello existe el comando `git rm`, que además elimina el archivo de tu directorio de trabajo de manera que no aparezca la próxima vez como un archivo no rastreado.

Si simplemente eliminas el archivo de tu directorio de trabajo, aparecerá en la sección “Changes not staged for commit” (esto es, *sin preparar*) en la salida de `git status`:

```
$ 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")
```

Ahora, si ejecutas `git rm`, entonces se prepara la eliminación del archivo:

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

    deleted:    PROJECTS.md
```

Con la próxima confirmación, el archivo habrá desaparecido y no volverá a ser rastreado. Si modificaste el archivo y ya lo habías añadido al índice, tendrás que forzar su eliminación con la opción `-f`. Esta propiedad existe por seguridad, para prevenir que elimines accidentalmente datos que aun no han sido guardados como una instantánea y que por lo tanto no podrás recuperar luego con Git.

Otra cosa que puedes querer hacer es mantener el archivo en tu directorio de trabajo pero eliminarlo del área de preparación. En otras palabras, quisieras mantener el archivo en tu disco duro pero sin que Git lo siga rastreando. Esto puede ser particularmente útil si olvidaste añadir algo en tu archivo `.gitignore` y lo preparaste accidentalmente, algo como un gran archivo de trazas a un montón de archivos compilados `.a`. Para hacerlo, utiliza la opción `--cached`:

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

Al comando `git rm` puedes pasarle archivos, directorios y patrones glob. Lo que significa que puedes hacer cosas como

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

Fíjate en la barra invertida (\) antes del asterisco *. Esto es necesario porque Git hace su propia expansión de nombres de archivo, aparte de la expansión hecha por tu terminal. Este comando elimina todos los archivo que tengan la extensión .log dentro del directorio log/. O también puedes hacer algo como:

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

Este comando elimina todos los archivos que acaben con ~.

Cambiar el Nombre de los Archivos

Al contrario que muchos sistemas VCS, Git no rastrea explícitamente los cambios de nombre en archivos. Si renombras un archivo en Git, no se guardará ningún metadato que indique que renombraste el archivo. Sin embargo, Git es bastante listo como para detectar estos cambios luego que los has hecho - más adelante, veremos cómo se detecta el cambio de nombre.

Por esto, resulta confuso que Git tenga un comando `mv`. Si quieres renombrar un archivo en Git, puedes ejecutar algo como

```
$ git mv file_from file_to
```

y funcionará bien. De hecho, si ejecutas algo como eso y ves el estatus, verás que Git lo considera como un renombramiento de archivo:

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

    renamed:    README.md -> README
```

Sin embargo, eso es equivalente a ejecutar algo como esto:

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

Git se da cuenta que es un renombramiento implícito, así que no importa si renombras el archivo de esa manera o a través del comando `mv`. La única diferencia real es que `mv` es un solo comando en vez de tres - existe por conveniencia. De hecho, puedes usar la herramienta que quieras para renombrar un archivo y luego realizar el proceso `rm/add` antes de confirmar.

Ver el Historial de Confirmaciones

Después de haber hecho varias confirmaciones, o si has clonado un repositorio que ya tenía un histórico de confirmaciones, probablemente quieras mirar atrás para ver qué modificaciones se han llevado a cabo. La herramienta más básica y potente para hacer esto es el comando `git log`.

Estos ejemplos usan un proyecto muy sencillo llamado “simplegit”. Para clonar el proyecto, ejecuta:

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

Cuando ejecutes `git log` sobre este proyecto, deberías ver una salida similar a esta:

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

    changed the version number

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

    removed unnecessary test

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

first commit

Por defecto, si no pasas ningún parámetro, `git log` lista las confirmaciones hechas sobre ese repositorio en orden cronológico inverso. Es decir, las confirmaciones más recientes se muestran al principio. Como puedes ver, este comando lista cada confirmación con su suma de comprobación SHA-1, el nombre y dirección de correo del autor, la fecha y el mensaje de confirmación.

El comando `git log` proporciona gran cantidad de opciones para mostrarte exactamente lo que buscas. Aquí veremos algunas de las más usadas.

Una de las opciones más útiles es `-p`, que muestra las diferencias introducidas en cada confirmación. También puedes usar la opción `-2`, que hace que se muestren únicamente las dos últimas entradas del historial:

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

        changed the 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

        removed 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
\ No newline at end of file

```

Esta opción muestra la misma información, pero añadiendo tras cada entrada las diferencias que le corresponden. Esto resulta muy útil para revisiones de código, o para visualizar rápidamente lo que ha pasado en las confirmaciones enviadas por un colaborador. También puedes usar con `git log` una serie de opciones de resumen. Por ejemplo, si quieras ver algunas estadísticas de cada confirmación, puedes usar la opción `--stat`:

```

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

        changed the 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

        removed 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

        first commit

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

```

Como puedes ver, la opción `--stat` imprime tras cada confirmación una lista de archivos modificados, indicando cuántos han sido modificados y cuántas líneas han sido añadidas y eliminadas para cada uno de ellos, y un resumen de toda esta información.

Otra opción realmente útil es `--pretty`, que modifica el formato de la salida. Tienes unos cuantos estilos disponibles. La opción `oneline` imprime cada confirmación en una única línea, lo que puede resultar útil si estás analizando gran cantidad de confirmaciones. Otras opciones son `short`, `full` y `fuller`, que muestran la salida en un formato parecido, pero añadiendo menos o más información, respectivamente:

```
$ git log --pretty=oneline
ca82a6dff817ec66f44342007202690a93763949 changed the version number
085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 removed unnecessary test
a11bef06a3f659402fe7563abf99ad00de2209e6 first commit
```

La opción más interesante es `format`, que te permite especificar tu propio formato. Esto resulta especialmente útil si estás generando una salida para que sea analizada por otro programa —como específicas el formato explícitamente, sabes que no cambiará en futuras actualizaciones de Git—:

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

Table 2-1 lista algunas de las opciones más útiles aceptadas por `format`.

TABLE 2-1. Opciones útiles de `git log --pretty=format`

Opción	Descripción de la salida
<code>%H</code>	Hash de la confirmación
<code>%h</code>	Hash de la confirmación abreviado
<code>%T</code>	Hash del árbol
<code>%t</code>	Hash del árbol abreviado
<code>%P</code>	Hashes de las confirmaciones padre
<code>%p</code>	Hashes de las confirmaciones padre abreviados

Opción	Descripción de la salida
%an	Nombre del autor
%ae	Dirección de correo del autor
%ad	Fecha de autoría (el formato respeta la opción --date)
%ar	Fecha de autoría, relativa
%cn	Nombre del confirmador
%ce	Dirección de correo del confirmador
%cd	Fecha de confirmación
%cr	Fecha de confirmación, relativa
%s	Asunto

Puede que te estés preguntando la diferencia entre *autor* (*author*) y *confirmador* (*committer*). El autor es la persona que escribió originalmente el trabajo, mientras que el confirmador es quien lo aplicó. Por tanto, si mandas un parche a un proyecto, y uno de sus miembros lo aplica, ambos recibiréis reconocimiento —tú como autor, y el miembro del proyecto como confirmador—. Veremos esta distinción en mayor profundidad en [Chapter 5](#).

Las opciones `oneline` y `format` son especialmente útiles combinadas con otra opción llamada `--graph`. Ésta añade un pequeño gráfico ASCII mostrando tu historial de ramificaciones y uniones:

```
$ git log --pretty=format:"%h %s" --graph
* 2d3acf9 ignore errors from SIGCHLD on trap
* 5e3ee11 Merge branch 'master' of git://github.com/dustin/grit
  \
  | * 420eac9 Added a 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
```

Este tipo de salidas serán más interesantes cuando empecemos a hablar sobre ramificaciones y combinaciones en el próximo capítulo.

Éstas son sólo algunas de las opciones para formatear la salida de `git log` —existen muchas más. [Table 2-2](#) lista las opciones vistas hasta ahora, y algu-

nas otras opciones de formateo que pueden resultarte útiles, así como su efecto sobre la salida.

TABLE 2-2. Opciones típicas de *git log*

Opción	Descripción
-p	Muestra el parche introducido en cada confirmación.
--stat	Muestra estadísticas sobre los archivos modificados en cada confirmación.
--shortstat	Muestra solamente la línea de resumen de la opción --stat.
--name-only	Muestra la lista de archivos afectados.
--name-status	Muestra la lista de archivos afectados, indicando además si fueron añadidos, modificados o eliminados.
--abbrev-commit	Muestra solamente los primeros caracteres de la suma SHA-1, en vez de los 40 caracteres de que se compone.
--relative-date	Muestra la fecha en formato relativo (por ejemplo, “2 weeks ago” (“hace 2 semanas”)) en lugar del formato completo.
--graph	Muestra un gráfico ASCII con la historia de ramificaciones y uniones.
--pretty	Muestra las confirmaciones usando un formato alternativo. Posibles opciones son oneline, short, full, fuller y format (mediante el cual puedes especificar tu propio formato).

Limitar la Salida del Historial

Además de las opciones de formateo, `git log` acepta una serie de opciones para limitar su salida —es decir, opciones que te permiten mostrar únicamente parte de las confirmaciones—. Ya has visto una de ellas, la opción `-2`, que muestra sólo las dos últimas confirmaciones. De hecho, puedes hacer `-<n>`, siendo `n` cualquier entero, para mostrar las últimas `n` confirmaciones. En realidad es poco probable que uses esto con frecuencia, ya que Git por defecto pagaña su salida para que veas cada página del historial por separado.

Sin embargo, las opciones temporales como `--since` (desde) y `--until` (hasta) sí que resultan muy útiles. Por ejemplo, este comando lista todas las confirmaciones hechas durante las dos últimas semanas:

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

Este comando acepta muchos formatos. Puedes indicar una fecha concreta ("2008-01-15"), o relativa, como "2 years 1 day 3 minutes ago" ("hace 2 años, 1 día y 3 minutos").

También puedes filtrar la lista para que muestre sólo aquellas confirmaciones que cumplen ciertos criterios. La opción `--author` te permite filtrar por autor, y `--grep` te permite buscar palabras clave entre los mensajes de confirmación. (Ten en cuenta que si quieras aplicar ambas opciones simultáneamente, tienes que añadir `--all-match`, o el comando mostrará las confirmaciones que cumplan cualquiera de las dos, no necesariamente las dos a la vez.)

Otra opción útil es `-S`, la cual recibe una cadena y solo muestra las confirmaciones que cambiaron el código añadiendo o eliminando la cadena. Por ejemplo, si quieras encontrar la última confirmación que añadió o eliminó una referencia a una función específica, puede ejecutar:

```
$ git log -Sfunction_name
```

La última opción verdaderamente útil para filtrar la salida de `git log` es especificar una ruta. Si especificas la ruta de un directorio o archivo, puedes limitar la salida a aquellas confirmaciones que introdujeron un cambio en dichos archivos. Ésta debe ser siempre la última opción, y suele ir precedida de dos guiones (`--`) para separar la ruta del resto de opciones.

En **Table 2-3** se listan estas opciones, y algunas otras bastante comunes, a modo de referencia.

TABLE 2-3. Opciones para limitar la salida de `git log`

Opción	Descripción
<code>-(n)</code>	Muestra solamente las últimas n confirmaciones
<code>--since, --after</code>	Muestra aquellas confirmaciones hechas después de la fecha especificada.
<code>--until, --before</code>	Muestra aquellas confirmaciones hechas antes de la fecha especificada.
<code>--author</code>	Muestra solo aquellas confirmaciones cuyo autor coincide con la cadena especificada.
<code>--committer</code>	Muestra solo aquellas confirmaciones cuyo comitidor coincide con la cadena especificada.
<code>-S</code>	Muestra solo aquellas confirmaciones que añadan o eliminan código que corresponda con la cadena especificada.

Por ejemplo, si quieras ver cuáles de las confirmaciones hechas sobre archivos de prueba del código fuente de Git fueron enviadas por Junio Hamano, y no fueron uniones, en el mes de octubre de 2008, ejecutarías algo así:

```
$ git log --pretty="%h - %s" --author=gitster --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
```

De las casi 40.000 confirmaciones en la historia del código fuente de Git, este comando muestra las 6 que cumplen estas condiciones.

Deshacer Cosas

En cualquier momento puede que quieras deshacer algo. Aquí repasaremos algunas herramientas básicas usadas para deshacer cambios que hayas hecho. Ten cuidado, a veces no es posible recuperar algo luego que lo has deshecho. Esta es una de las pocas áreas en las que Git puede perder parte de tu trabajo si cometes un error.

Uno de las acciones más comunes a deshacer es cuando confirmas un cambio antes de tiempo y olvidas agregar algún archivo, o te equivocas en el mensaje de confirmación. Si quieras rehacer la confirmación, puedes reconfirmar con la opción `--amend`:

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

Este comando utiliza tu área de preparación para la confirmación. Si no has hecho cambios desde tu última confirmación (por ejemplo, ejecutas este comando justo después de tu confirmación anterior), entonces la instantánea lucirá exactamente igual, y lo único que cambiarás será el mensaje de confirmación.

Se lanzará el mismo editor de confirmación, pero verás que ya incluye el mensaje de tu confirmación anterior. Puedes editar el mensaje como siempre y se sobreescibirá tu confirmación anterior.

Por ejemplo, si confirmas y luego te das cuenta que olvidaste preparar los cambios de un archivo que querías incluir en esta confirmación, puedes hacer lo siguiente:

```
$ git commit -m 'initial commit'
$ git add forgotten_file
$ git commit --amend
```

Al final terminarás con una sola confirmación - la segunda confirmación reemplaza el resultado de la primera.

Deshacer un Archivo Preparado

Las siguientes dos secciones demuestran cómo lidiar con los cambios de tu área de preparación y tú directorio de trabajo. Afortunadamente, el comando que usas para determinar el estado de esas dos áreas también te recuerda cómo deshacer los cambios en ellas. Por ejemplo, supongamos que has cambiado dos archivos y que quieras confirmarlos como dos cambios separados, pero accidentalmente has escrito `git add *` y has preparado ambos. ¿Cómo puedes sacar del área de preparación uno de ellos? El comando `git status` te recuerda cómo:

```
$ 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
```

Justo debajo del texto “Changes to be committed” (“Cambios a ser confirmados”, en inglés), verás que dice que uses `git reset HEAD <file>...` para deshacer la preparación. Por lo tanto, usemos el consejo para deshacer la preparación del archivo `CONTRIBUTING.md`:

```
$ 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
```

El comando es un poco raro, pero funciona. El archivo `CONTRIBUTING.md` esta modificado y, nuevamente, no preparado.

A pesar de que `git reset` *puede* ser un comando peligroso si lo llamas con `--hard`, en este caso el archivo que está en tu directorio de trabajo no se toca. Ejecutar `git reset` sin opciones no es peligroso – solo toca el área de preparación.

Por ahora lo único que necesitas saber sobre el comando `git reset` es esta invocación mágica. Entraremos en mucho más detalle sobre qué hace `reset` y como dominarlo para que haga cosas realmente interesantes en “**Reset Demystified**”.

Deshacer un Archivo Modificado

¿Qué tal si te das cuenta que no quieres mantener los cambios del archivo `CONTRIBUTING.md`? ¿Cómo puedes restaurarlo fácilmente - volver al estado en el que estaba en la última confirmación (o cuando estaba recién clonado, o como sea que haya llegado a tu directorio de trabajo)? Afortunadamente, `git status` también te dice cómo hacerlo. En la salida anterior, el área no preparada lucía así:

```
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
```

Allí se te indica explícitamente como descartar los cambios que has hecho. Hagamos lo que nos dice:

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

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

Ahora puedes ver que los cambios se han revertido.

Es importante entender que `git checkout -- [archivo]` es un comando peligroso. Cualquier cambio que le hayas hecho a ese archivo desaparecerá – acabas de sobreescribirlo con otro archivo. Nunca utilices este comando a menos que estés absolutamente seguro de que ya no quieras el archivo.

Para mantener los cambios que has hecho y a la vez deshacerte del archivo temporalmente, hablaremos sobre cómo esconder archivos (*stashing*, en inglés) y sobre ramas en **Chapter 3**; normalmente, estas son las mejores maneras de hacerlo.

Recuerda, todo lo que esté *confirmado* en Git puede recuperarse. Incluso *commits* que estuvieron en ramas que han sido eliminadas o *commits* que fueron sobreescritos con `--amend` pueden recuperarse (véase “**Data Recovery**” para recuperación de datos). Sin embargo, es posible que no vuelvas a ver jamás cualquier cosa que pierdas y que nunca haya sido confirmada.

Trabajar con Remotos

Para poder colaborar en cualquier proyecto Git, necesitas saber cómo gestionar repositorios remotos. Los repositorios remotos son versiones de tu proyecto que están hospedadas en Internet en cualquier otra red. Puedes tener varios de ellos, y en cada uno tendrás generalmente permisos de solo lectura o de lectura y escritura. Colaborar con otras personas implica gestionar estos repositorios remotos y enviar y traer datos de ellos cada vez que necesites compartir tu trabajo. Gestionar repositorios remotos incluye saber cómo añadir un repositorio remoto, eliminar los remotos que ya no son válidos, gestionar varias ramas remotas y definir si deben rastrearse o no, y más. En esta sección, trataremos algunas de estas habilidades de gestión de remotos.

Ver Tus Remotos

Para ver los remotos que tienes configurados, debes ejecutar el comando `git remote`. Mostrará los nombres de cada uno de los remotos que tienes especificados. Si has clonado tu repositorio, deberías ver al menos `origin` (origen, en inglés) – este es el nombre que por defecto Git le da al servidor del que has clonado:

```
$ 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
```

También puedes pasar la opción `-v`, la cual muestra las URLs que Git ha asociado al nombre y que serán usadas al leer y escribir en ese remoto:

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

Si tienes más de un remoto, el comando los listará todos. Por ejemplo, un repositorio con múltiples remotos para trabajar con distintos colaboradores podría verse de la siguiente manera.

```
$ 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)
```

Esto significa que podemos traer contribuciones de cualquiera de estos usuarios fácilmente. Es posible que también tengamos permisos para enviar datos a algunos, aunque no podemos saberlo desde aquí.

Fíjate que estos remotos usan distintos protocolos; hablaremos sobre ello más adelante, en “**Configurando Git en un servidor**”.

Añadir Repositorios Remotos

En secciones anteriores hemos mencionado y dado alguna demostración de cómo añadir repositorios remotos. Ahora veremos explícitamente cómo hacerlo. Para añadir un remoto nuevo y asociarlo a un nombre que puedas referenciar fácilmente, ejecuta `git remote add [nombre] [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)
```

A partir de ahora puedes usar el nombre `pb` en la línea de comandos en lugar de la URL entera. Por ejemplo, si quieres traer toda la información que tiene Paul pero tú aun no tienes en tu repositorio, puedes ejecutar `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
```

La rama maestra de Paul ahora es accesible localmente con el nombre `pb/master` - puedes combinarla con alguna de tus ramas, o puedes crear una rama local en ese punto si quieres inspeccionarla. (Hablaremos con más detalle acerca de qué son las ramas y cómo utilizarlas en [Chapter 3](#).)

Traer y Combinar Remotos

Como hemos visto hasta ahora, para obtener datos de tus proyectos remotos puedes ejecutar:

```
$ git fetch [remote-name]
```

El comando irá al proyecto remoto y se traerá todos los datos que aun no tienes de dicho remoto. Luego de hacer esto, tendrás referencias a todas las ramas del remoto, las cuales puedes combinar e inspeccionar cuando quieras.

Si clonas un repositorio, el comando de clonar automáticamente añade ese repositorio remoto con el nombre “origin”. Por lo tanto, `git fetch origin` se trae todo el trabajo nuevo que ha sido enviado a ese servidor desde que lo clonaste (o desde la última vez que trajiste datos). Es importante destacar que el comando `git fetch` solo trae datos a tu repositorio local - ni lo combina automáticamente con tu trabajo ni modifica el trabajo que llevas hecho. La combinación con tu trabajo debes hacerla manualmente cuando estés listo.

Si has configurado una rama para que rastree una rama remota (más información en la siguiente sección y en [Chapter 3](#)), puedes usar el comando `git pull` para traer y combinar automáticamente la rama remota con tu rama actual. Es posible que este sea un flujo de trabajo mucho más cómodo y fácil para ti; y por defecto, el comando `git clone` le indica automáticamente a tu rama maestra local que rastree la rama maestra remota (o como se llame la rama por defecto) del servidor del que has clonado. Generalmente, al ejecutar `git pull` traerás datos del servidor del que clonaste originalmente y se intentará combinar automáticamente la información con el código en el que estás trabajando.

Enviar a Tus Remotos

Cuando tienes un proyecto que quieres compartir, debes enviarlo a un servidor. El comando para hacerlo es simple: `git push [nombre-remoto] [nombre-rama]`. Si quieres enviar tu rama `master` a tu servidor `origin` (recuerda, clonar un repositorio establece esos nombres automáticamente), entonces puedes ejecutar el siguiente comando y se enviarán todos los *commits* que hayas hecho al servidor:

```
$ git push origin master
```

Este comando solo funciona si clonaste de un servidor sobre el que tienes permisos de escritura y si nadie más ha enviado datos por el medio. Si alguien más clona el mismo repositorio que tú y envía información antes que tú, tu envío será rechazado. Tendrás que traerte su trabajo y combinarlo con el tuyo antes de que puedas enviar datos al servidor. Para información más detallada sobre cómo enviar datos a servidores remotos, véase [Chapter 3](#).

Inspeccionar un Remoto

Si quieres ver más información acerca de un remoto en particular, puedes ejecutar el comando `git remote show [nombre-remoto]`. Si ejecutas el comando con un nombre en particular, como `origin`, verás algo como lo siguiente:

```
$ 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)
```

El comando lista la URL del repositorio remoto y la información del rastreo de ramas. El comando te indica claramente que si estás en la rama maestra y ejecutas el comando `git pull`, automáticamente combinará la rama maestra remota luego de haber traído toda la información de ella. También lista todas las referencias remotas de las que ha traído datos.

Ejemplos como este son los que te encontrarás normalmente. Sin embargo, si usas Git de forma más avanzada, puede que obtengas mucha más información de un `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
  markdown-strip               pushes to markdown-strip
  master                       pushes to master
```

(up to date)
 (up to date)
 (up to date)

Este comando te indica a cuál rama enviarás información automáticamente cada vez que ejecutas `git push`, dependiendo de la rama en la que estés. También te muestra cuáles ramas remotas no tienes aun, cuáles ramas remotas tienes que han sido eliminadas del servidor, y varias ramas que serán combinadas automáticamente cuando ejecutes `git pull`.

Eliminar y Renombrar Remotos

Si quieres cambiar el nombre de la referencia de un remoto puedes ejecutar `git remote rename`. Por ejemplo, si quieres cambiar el nombre de `pb` a `paul`, puedes hacerlo con `git remote rename`:

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

Es importante destacar que al hacer esto también cambias el nombre de las ramas remotas. Por lo tanto, lo que antes estaba referenciado como `pb/master` ahora lo está como `paul/master`.

Si por alguna razón quieres eliminar un remoto - has cambiado de servidor o no quieres seguir utilizando un *mirror*, o quizás un colaborador a dejado de trabajar en el proyecto - puedes usar `git remote rm`:

```
$ git remote rm paul
$ git remote
origin
```

Etiquetado

Como muchos VCS, Git tiene la posibilidad de etiquetar puntos específicos del historial como importantes. Esta funcionalidad se usa típicamente para marcar versiones de lanzamiento (v1.0, por ejemplo). En esta sección, aprenderás cómo listar las etiquetas disponibles, cómo crear nuevas etiquetas y cuáles son los distintos tipos de etiquetas.

Listar Tus Etiquetas

Listar las etiquetas disponibles en Git es sencillo. Simplemente escribe `git tag`:

```
$ git tag
v0.1
v1.3
```

Este comando lista las etiquetas en orden alfabético; el orden en el que aparecen no tiene mayor importancia.

También puedes buscar etiquetas con un patrón particular. El repositorio del código fuente de Git, por ejemplo, contiene más de 500 etiquetas. Si solo te interesa ver la serie 1.8.5, puedes ejecutar:

```
$ 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
```

Crear Etiquetas

Git utiliza dos tipos principales de etiquetas: *ligeras* y *anotadas*.

Una etiqueta ligera es muy parecido a una rama que no cambia - simplemente es un puntero a un *commit* específico.

Sin embargo, las etiquetas anotadas se guardan en la base de datos de Git como objetos enteros. Tienen un *checksum*; contienen el nombre del etiquetador, correo electrónico y fecha; tienen un mensaje asociado; y pueden ser firmadas y verificadas con *GNU Privacy Guard* (GPG). Normalmente se recomienda que crees etiquetas anotadas, de manera que tengas toda esta información; pero si quieras una etiqueta temporal o por alguna razón no estás interesado en esa información, entonces puedes usar las etiquetas ligeras.

Etiquetas Anotadas

Crear una etiqueta anotada en Git es sencillo. La forma más fácil de hacer es especificar la opción `-a` cuando ejecutas el comando `tag`:

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

La opción `-m` especifica el mensaje de la etiqueta, el cual es guardado junto con ella. Si no especificas el mensaje de una etiqueta anotada, Git abrirá el editor de texto para que lo escribas.

Puedes ver la información de la etiqueta junto con el *commit* que está etiquetado al usar el comando `git show`:

```
$ 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 <scchacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

        changed the version number
```

El comando muestra la información del etiquetador, la fecha en la que el *commit* fue etiquetado y el mensaje de la etiqueta, antes de mostrar la información del *commit*.

Etiquetas Ligeras

La otra forma de etiquetar un *commit* es mediante una etiqueta ligera. Una etiqueta ligera no es más que el *checksum* de un *commit* guardado en un archivo - no incluye más información. Para crear una etiqueta ligera, no pases las opciones `-a`, `-s` ni `-m`:

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

Esta vez, si ejecutas `git show` sobre la etiqueta, no verás la información adicional. El comando solo mostrará el *commit*:

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

        changed the version number
```

Etiquetado Tardío

También puedes etiquetar *commits* mucho tiempo después de haberlos hecho. Supongamos que tu historial luce como el siguiente:

```
$ git log --pretty=oneline
15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'
a6b4c97498bd301d84096da251c98a07c7723e65 beginning write support
0d52aaab4479697da7686c15f77a3d64d9165190 one more thing
6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'
0b7434d86859cc7b8c3d5e1dddfe66ff742fcbe added a commit function
4682c3261057305bdd616e23b64b0857d832627b added a todo file
166ae0c4d3f420721acbb115cc33848dfcc2121a started write support
9fcfb02d0ae598e95dc970b74767f19372d61af8 updated rakefile
964f16d36dfccde844893cac5b347e7b3d44abbc commit the todo
8a5cbc430f1a9c3d00faaeffd07798508422908a updated readme
```

Ahora, supongamos que olvidaste etiquetar el proyecto en su versión v1.2, la cual corresponde al *commit* “updated rakefile”. Igual puedes etiquetarlo. Para etiquetar un *commit*, debes especificar el *checksum* del *commit* (o parte de él) al final del comando:

```
$ git tag -a v1.2 9fcfb02
```

Puedes ver que has etiquetado el 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 9fce02d0ae598e95dc970b74767f19372d61af8
Author: Magnus Chacon <mchacon@gee-mail.com>
Date:   Sun Apr 27 20:43:35 2008 -0700

        updated rakefile
        ...
...
```

Compartir Etiquetas

Por defecto, el comando `git push` no transfiere las etiquetas a los servidores remotos. Debes enviar las etiquetas de forma explícita al servidor luego de que las hayas creado. Este proceso es similar al de compartir ramas remotas - puede ejecutarse `git push origin [etiqueta]`.

```
$ 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
```

Si quieres enviar varias etiquetas a la vez, puedes usar la opción `--tags` del comando `git push`. Esto enviará al servidor remoto todas las etiquetas que aun no existen en él.

```
$ 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
```

Por lo tanto, cuando alguien clone o traiga información de tu repositorio, también obtendrá todas las etiquetas.

Sacar una Etiqueta

En Git, no puedes sacar (*check out*) una etiqueta, pues no es algo que puedas mover. Si quieres colocar en tu directorio de trabajo una versión de tu repositorio que coincida con alguna etiqueta, debes crear una rama nueva en esa etiqueta:

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

Obviamente, si haces esto y luego confirmas tus cambios, tu rama `version2` será ligeramente distinta a tu etiqueta `v2.0.0` puesto que incluirá tus nuevos cambios; así que ten cuidado.

Alias de Git

Antes de terminar este capítulo sobre fundamentos de Git, hay otro pequeño consejo que puede hacer que tu experiencia con Git sea más simple, sencilla y familiar: los alias. No volveremos a mencionarlos más adelante en este libro, ni supondremos que los has utilizado, pero probablemente deberías saber cómo utilizarlos.

Git no deduce automáticamente tu comando si lo tecleas parcialmente. Si no quieres teclear el nombre completo de cada comando de Git, puedes establecer fácilmente un alias para cada comando mediante `git config`. Aquí tienes algunos ejemplos que te pueden interesar:

```
$ git config --global alias.co checkout
$ git config --global alias.br branch
```

```
$ git config --global alias.ci commit
$ git config --global alias.st status
```

Esto significa que, por ejemplo, en lugar de teclear `git commit`, solo necesitas teclear `git ci`. A medida que uses Git, probablemente también utilizarás otros comandos con frecuencia; no dudes en crear nuevos alias.

Esta técnica también puede resultar útil para crear comandos que en tu opinión deberían existir. Por ejemplo, para corregir el problema de usabilidad que encontraste al quitar del área de preparación un archivo, puedes añadir tu propio alias a Git:

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

Esto hace que los dos comandos siguientes sean equivalentes:

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

Esto parece un poco más claro. También es frecuente añadir un comando `last`, de este modo:

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

De esta manera, puedes ver fácilmente cuál fue la última confirmación:

```
$ 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>
```

Como puedes ver, Git simplemente sustituye el nuevo comando por lo que sea que hayas puesto en el alias. Sin embargo, quizás quieras ejecutar un comando externo, en lugar de un subcomando de Git. En ese caso, puedes comenzar el comando con un carácter `!`. Esto resulta útil si escribes tus propias

herramientas para trabajar con un repositorio de Git. Podemos demostrarlo creando el alias `git visual` para ejecutar `gitk`:

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

Resumen

En este momento puedes hacer todas las operaciones básicas de Git a nivel local: Crear o clonar un repositorio, hacer cambios, preparar y confirmar esos cambios y ver la historia de los cambios en el repositorio. A continuación cubriremos la mejor característica de Git: Su modelo de ramas.

3

Ramificaciones en Git

Cualquier sistema de control de versiones moderno tiene algún mecanismo para soportar distintos ramales. Cuando hablamos de ramificaciones, significa que tú has tomado la rama principal de desarrollo (master) y a partir de ahí has continuado trabajando sin seguir la rama principal de desarrollo. En muchas sistemas de control de versiones este proceso es costoso, pues a menudo requiere crear una nueva copia del código, lo cual puede tomar mucho tiempo cuando se trata de proyectos grandes.

Algunas personas resaltan que uno de los puntos más fuertes de Git es su sistema de ramificaciones y lo cierto es que esto le hace resaltar sobre los otros sistemas de control de versiones. ¿Por qué esto es tan importante? La forma en la que Git maneja las ramificaciones es increíblemente rápida, haciendo así de las operaciones de ramificación algo casi instantáneo, al igual que el avance o el retroceso entre distintas ramas, lo cual también es tremadamente rápido. A diferencia de otros sistemas de control de versiones, Git promueve un ciclo de desarrollo donde las ramas se crean y se unen ramas entre sí, incluso varias veces en el mismo día. Entender y manejar esta opción te proporciona una poderosa y exclusiva herramienta que puede, literalmente, cambiar la forma en la que desarrollas.

¿Qué es una rama?

Para entender realmente cómo ramifica Git, previamente hemos de examinar la forma en que almacena sus datos.

Recordando lo citado en **Chapter 1**, Git no los almacena de forma incremental (guardando solo diferencias), sino que los almacena como una serie de instantáneas (copias puntuales de los archivos completos, tal y como se encuentran en ese momento).

En cada confirmación de cambios (commit), Git almacena una instantánea de tu trabajo preparado. Dicha instantánea contiene además unos metadatos con el autor y el mensaje explicativo, y uno o varios apuntadores a las confir-

maciones (commit) que sean padres directos de esta (un parente en los casos de confirmación normal, y múltiples padres en los casos de estar confirmando una fusión (merge) de dos o más ramas).

Para ilustrar esto, vamos a suponer, por ejemplo, que tienes una carpeta con tres archivos, que preparas (stage) todos ellos y los confirmas (commit). Al preparar los archivos, Git realiza una suma de control de cada uno de ellos (un resumen SHA-1, tal y como se mencionaba en **Chapter 1**), almacena una copia de cada uno en el repositorio (estas copias se denominan “blobs”), y guarda cada suma de control en el área de preparación (staging area):

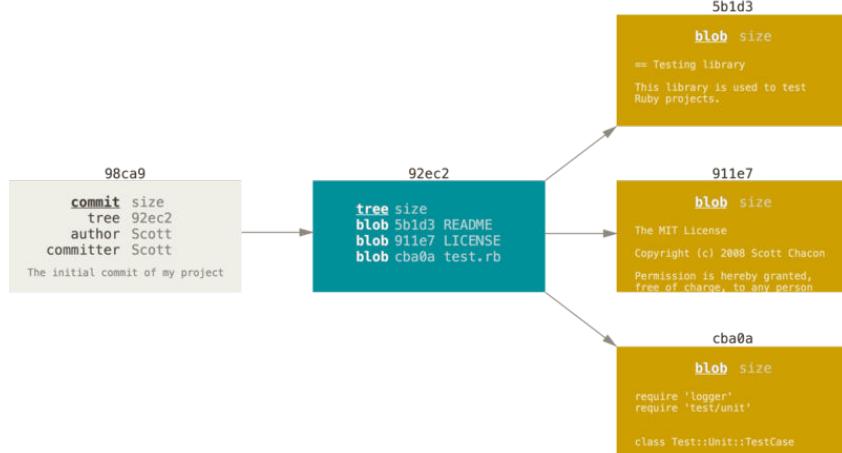
```
$ git add README test.rb LICENSE
$ git commit -m 'initial commit of my project'
```

Cuando creas una confirmación con el comando `git commit`, Git realiza sumas de control de cada subdirectorio (en el ejemplo, solamente tenemos el directorio principal del proyecto), y las guarda como objetos árbol en el repositorio Git. Después, Git crea un objeto de confirmación con los metadatos pertinentes y un apuntador al objeto árbol raíz del proyecto.

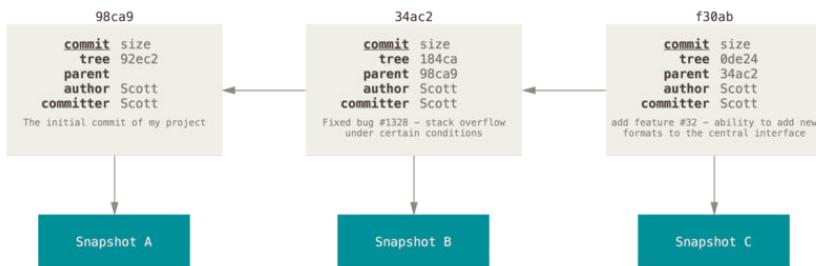
En este momento, el repositorio de Git contendrá cinco objetos: un “blob” para cada uno de los tres archivos, un árbol con la lista de contenidos del directorio (más sus respectivas relaciones con los “blobs”), y una confirmación de cambios (commit) apuntando a la raíz de ese árbol y conteniendo el resto de metadatos pertinentes.

FIGURE 3-1

Una confirmación y sus árboles



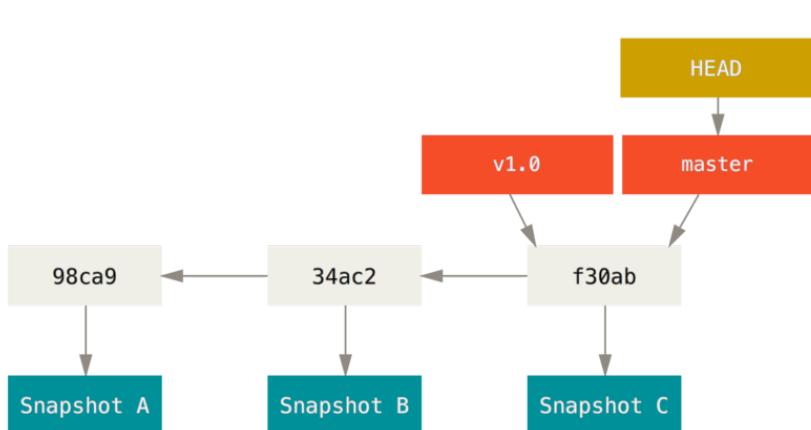
Si haces más cambios y vuelves a confirmar, la siguiente confirmación guardará un apuntador su confirmación precedente.

**FIGURE 3-2**

Confirmaciones y sus predecesoras

Una rama Git es simplemente un apuntador móvil apuntando a una de esas confirmaciones. La rama por defecto de Git es la rama `master`. Con la primera confirmación de cambios que realicemos, se creará esta rama principal `master` apuntando a dicha confirmación. En cada confirmación de cambios que realicemos, la rama irá avanzando automáticamente.

La rama “`master`” en Git no es una rama especial. Es como cualquier otra rama. La única razón por la cual aparece en casi todos los repositorios es porque es la que crea por defecto el comando `git init` y la gente no se molesta en cambiarle el nombre.

**FIGURE 3-3**

Una rama y su historial de confirmaciones

Crear una Rama Nueva

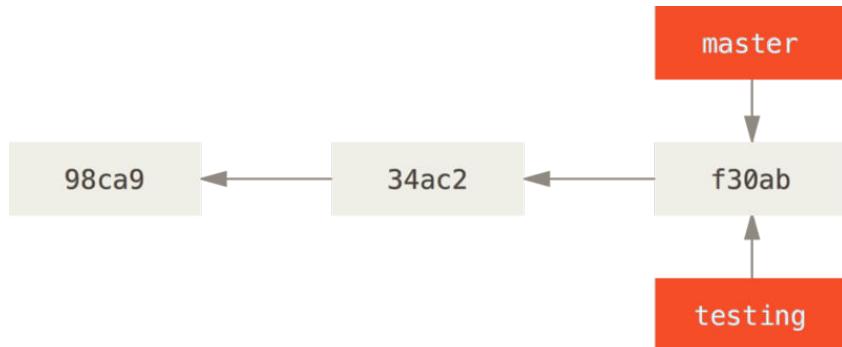
¿Qué sucede cuando creas una nueva rama? Bueno..., simplemente se crea un nuevo apuntador para que lo puedas mover libremente. Por ejemplo, supongamos que quieres crear una rama nueva denominada “testing”. Para ello, usarás el comando `git branch`:

```
$ git branch testing
```

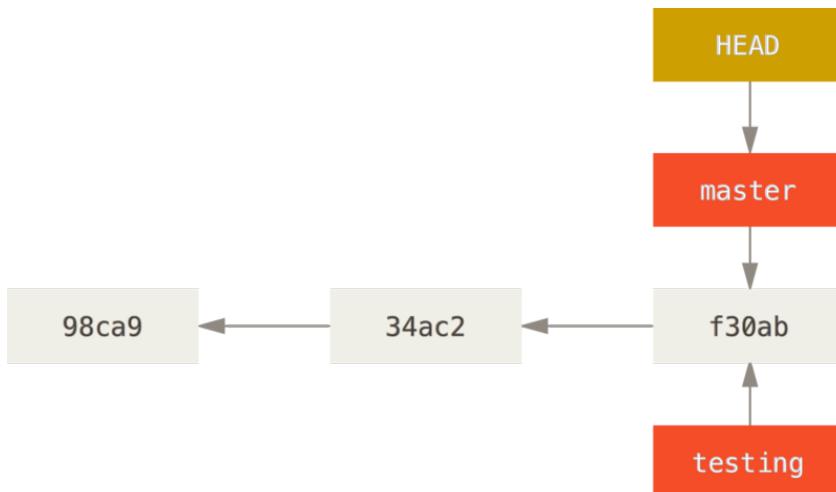
Esto creará un nuevo apuntador apuntando a la misma confirmación donde estés actualmente.

FIGURE 3-4

Dos ramas apuntando al mismo grupo de confirmaciones



Y, ¿cómo sabe Git en qué rama estás en este momento? Pues..., mediante un apuntador especial denominado HEAD. Aunque es preciso comentar que este HEAD es totalmente distinto al concepto de HEAD en otros sistemas de control de cambios como Subversion o CVS. En Git, es simplemente el apuntador a la rama local en la que tú estés en ese momento, en este caso la rama `master`; pues el comando `git branch` solamente crea una nueva rama, y no salta a dicha rama.

**FIGURE 3-5**

Apuntador HEAD a la rama donde estás actualmente

Esto puedes verlo fácilmente al ejecutar el comando `git log` para que te muestre a dónde apunta cada rama. Esta opción se llama `--decorate`.

```
$ git log --oneline --decorate
f30ab (HEAD, master, testing) add feature #32 - ability to add new
34ac2 fixed bug #1328 - stack overflow under certain conditions
98ca9 initial commit of my project
```

Puedes ver que las ramas “master” y “testing” están junto a la confirmación f30ab.

Cambiar de Rama

Para saltar de una rama a otra, tienes que utilizar el comando `git checkout`. Hagamos una prueba, saltando a la rama `testing` recién creada:

```
$ git checkout testing
```

Esto mueve el apuntador HEAD a la rama `testing`.

FIGURE 3-6

El apuntador HEAD apunta a la rama actual

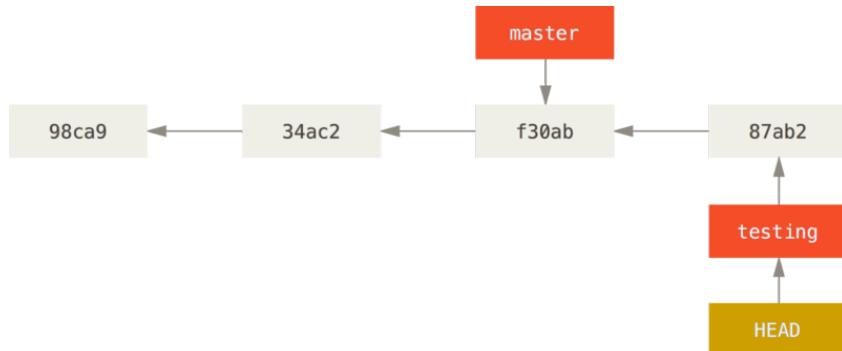


¿Cuál es el significado de todo esto? Bueno... lo veremos tras realizar otra confirmación de cambios:

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

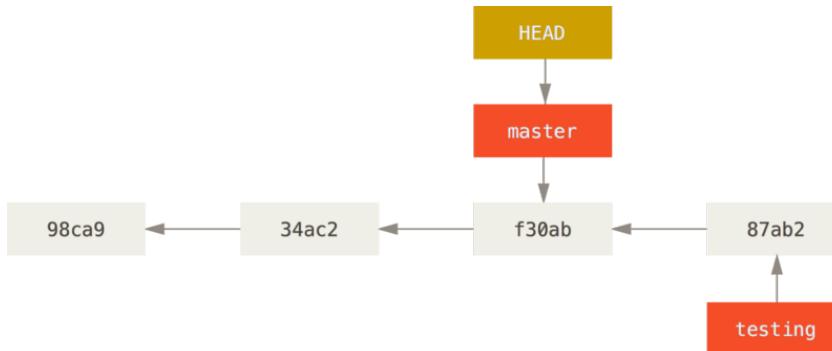
FIGURE 3-7

La rama apuntada por HEAD avanza con cada confirmación de cambios



Observamos algo interesante: la rama **testing** avanza, mientras que la rama **master** permanece en la confirmación donde estaba cuando lanzaste el comando `git checkout` para saltar. Volvamos ahora a la rama **master**:

```
$ git checkout master
```

**FIGURE 3-8**

HEAD apunta a otra rama cuando hacemos un salto

Este comando realiza dos acciones: Mueve el apuntador HEAD de nuevo a la rama `master`, y revierte los archivos de tu directorio de trabajo; dejándolos tal y como estaban en la última instantánea confirmada en dicha rama `master`. Esto supone que los cambios que hagas desde este momento en adelante divergirán de la antigua versión del proyecto. Básicamente, lo que se está haciendo es rebobinar el trabajo que habías hecho temporalmente en la rama `testing`; de tal forma que puedas avanzar en otra dirección diferente.

SALTAR ENTRE RAMAS CAMBIA ARCHIVOS EN TU DIRECTORIO DE TRABAJO

Es importante destacar que cuando saltas a una rama en Git, los archivos de tu directorio de trabajo cambian. Si saltas a una rama antigua, tu directorio de trabajo retrocederá para verse como lo hacia la última vez que confirmaste un cambio en dicha rama. Si Git no puede hacer el cambio limpiamente, no te dejará saltar.

Haz algunos cambios más y confírmalos:

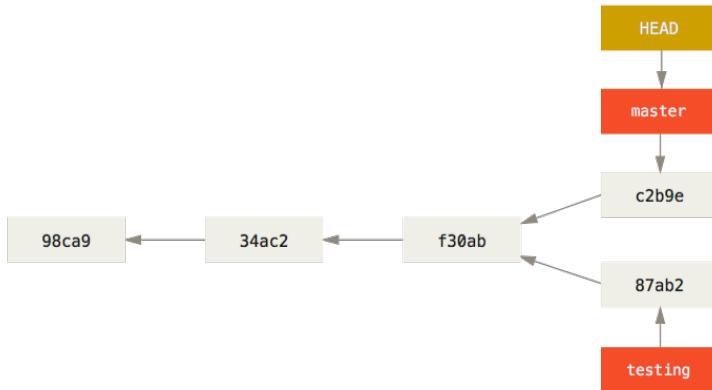
```
$ vim test.rb
$ git commit -a -m 'made other changes'
```

Ahora el historial de tu proyecto diverge (ver **Figure 3-9**). Has creado una rama y saltado a ella, has trabajado sobre ella; has vuelto a la rama original, y has trabajado también sobre ella. Los cambios realizados en ambas sesiones de trabajo están aislados en ramas independientes: puedes saltar libremente de

una a otra según estimes oportuno. Y todo ello simplemente con tres comandos: `git branch`, `git checkout` y `git commit`.

FIGURE 3-9

Los registros de las ramas divergen



También puedes ver esto fácilmente utilizando el comando `git log`. Si ejecutas `git log --oneline --decorate --graph --all` te mostrará el historial de tus confirmaciones, indicando dónde están los apuntadores de tus ramas y como ha divergido tu histórico.

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

Debido a que una rama Git es realmente un simple archivo que contiene los 40 caracteres de una suma de control SHA-1, (representando la confirmación de cambios a la que apunta), no cuesta nada el crear y destruir ramas en Git. Crear una nueva rama es tan rápido y simple como escribir 41 bytes en un archivo, (40 caracteres y un retorno de carro).

Esto contrasta fuertemente con los métodos de ramificación usados por otros sistemas de control de versiones, en los que crear una rama nueva su-

pone el copiar todos los archivos del proyecto a un directorio adicional nuevo. Esto puede llevar segundos o incluso minutos, dependiendo del tamaño del proyecto; mientras que en Git el proceso es siempre instantáneo. Y, además, debido a que se almacenan también los nodos padre para cada confirmación, el encontrar las bases adecuadas para realizar una fusión entre ramas es un proceso automático y generalmente sencillo de realizar. Animando así a los desarrolladores a utilizar ramificaciones frecuentemente.

Vamos a ver el por qué merece la pena hacerlo así.

Procedimientos Básicos para Ramificar y Fusionar

Vamos a presentar un ejemplo simple de ramificar y de fusionar, con un flujo de trabajo que se podría presentar en la realidad. Imagina que sigues los siguientes pasos:

1. Trabajas en un sitio web.
2. Creas una rama para un nuevo tema sobre el que quieres trabajar.
3. Realizas algo de trabajo en esa rama.

En este momento, recibes una llamada avisándote de un problema crítico que has de resolver. Y sigues los siguientes pasos:

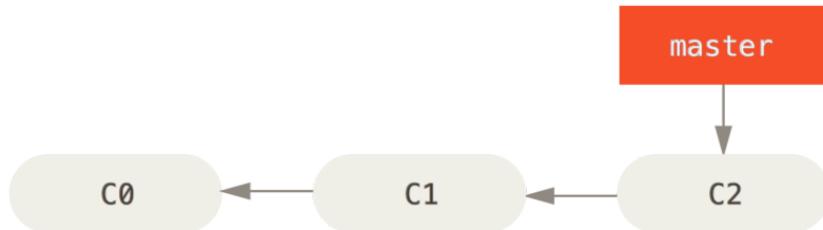
1. Vuelves a la rama de producción original.
2. Creas una nueva rama para el problema crítico y lo resuelves trabajando en ella.
3. Tras las pertinentes pruebas, fusionas (merge) esa rama y la envías (push) a la rama de producción.
4. Vuelves a la rama del tema en que andabas antes de la llamada y continúas tu trabajo.

Procedimientos Básicos de Ramificación

Imagina que estás trabajando en un proyecto y tienes un par de confirmaciones (commit) ya realizadas.

FIGURE 3-10

Un registro de confirmaciones corto y sencillo



Decides trabajar en el problema #53, según el sistema que tu compañía utiliza para llevar seguimiento de los problemas. Para crear una nueva rama y saltar a ella, en un solo paso, puedes utilizar el comando `git checkout` con la opción `-b`:

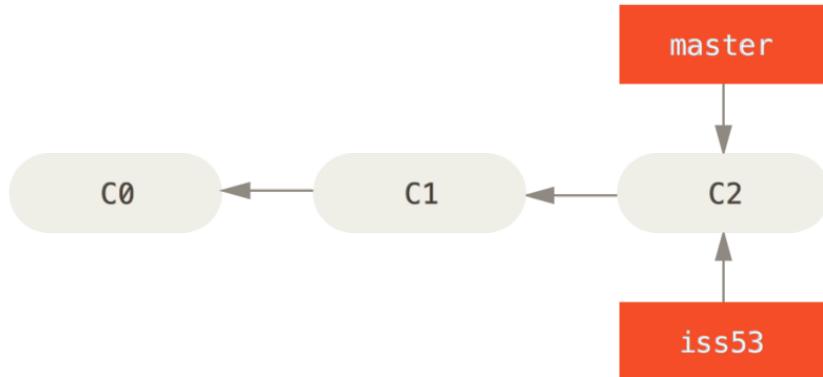
```
$ git checkout -b iss53
Switched to a new branch "iss53"
```

Esto es un atajo a:

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

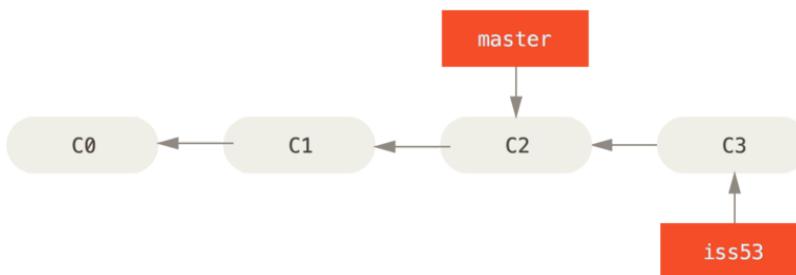
FIGURE 3-11

Crear un apuntador a la rama nueva



Trabajas en el sitio web y haces algunas confirmaciones de cambios (commits). Con ello avanzas la rama `iss53`, que es la que tienes activada (checked out) en este momento (es decir, a la que apunta HEAD):

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

**FIGURE 3-12**

La rama `iss53` ha avanzado con tu trabajo

Entonces, recibes una llamada avisándote de otro problema urgente en el sitio web y debes resolverlo inmediatamente. Al usar Git, no necesitas mezclar el nuevo problema con los cambios que ya habías realizado sobre el problema #53; ni tampoco perder tiempo revirtiendo esos cambios para poder trabajar sobre el contenido que está en producción. Basta con saltar de nuevo a la rama `master` y continuar trabajando a partir de allí.

Pero, antes de poder hacer eso, hemos de tener en cuenta que si tenemos cambios aún no confirmados en el directorio de trabajo o en el área de preparación, Git no nos permitirá saltar a otra rama con la que podríamos tener conflictos. Lo mejor es tener siempre un estado de trabajo limpio y despejado antes de saltar entre ramas. Y, para ello, tenemos algunos procedimientos (`stash` y corregir confirmaciones), que vamos a ver más adelante en “**Stashing and Cleaning**”. Por ahora, como tenemos confirmados todos los cambios, podemos saltar a la rama `master` sin problemas:

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

Tras esto, tendrás el directorio de trabajo exactamente igual a como estaba antes de comenzar a trabajar sobre el problema #53 y podrás concentrarte en el nuevo problema urgente. Es importante recordar que Git revierte el directorio

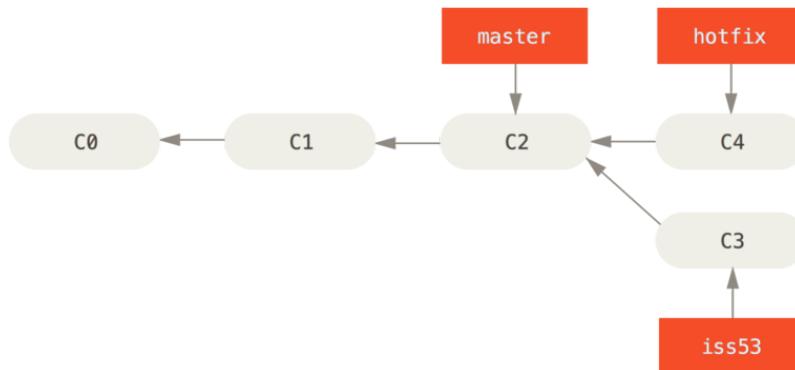
de trabajo exactamente al estado en que estaba en la confirmación (commit) apuntada por la rama que activamos (checkout) en cada momento. Git añade, quita y modifica archivos automáticamente para asegurar que tu copia de trabajo luce exactamente como lucía la rama en la última confirmación de cambios realizada sobre ella.

A continuación, es momento de resolver el problema urgente. Vamos a crear una nueva rama **hotfix**, sobre la que trabajar hasta resolverlo:

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

FIGURE 3-13

Rama **hotfix**
basada en la rama
master original

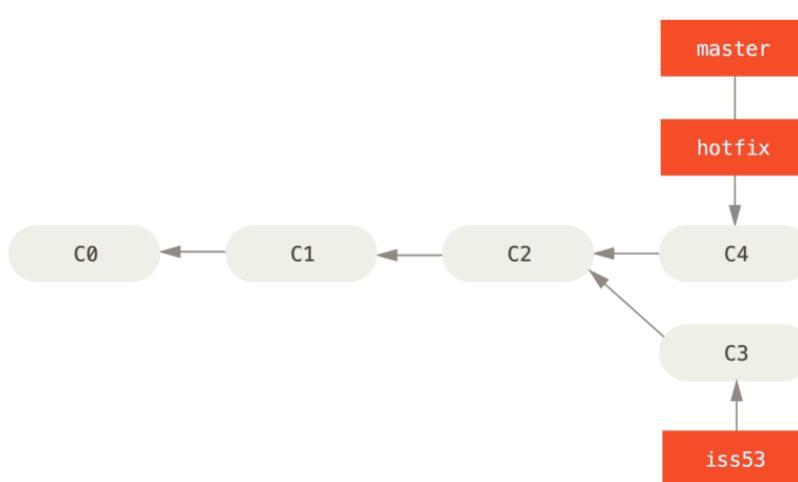


Puedes realizar las pruebas oportunas, asegurarte que la solución es correcta, e incorporar los cambios a la rama **master** para ponerlos en producción. Esto se hace con el comando **git merge**:

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

Notarás la frase “Fast forward” (“Avance rápido”, en inglés) que aparece en la salida del comando. Git ha movido el apuntador hacia adelante, ya que la confirmación apuntada en la rama donde has fusionado estaba directamente arriba respecto a la confirmación actual. Dicho de otro modo: cuando intentas fusionar una confirmación con otra confirmación accesible siguiendo directamente el historial de la primera; Git simplifica las cosas avanzando el puntero, ya que no hay ningún otro trabajo divergente a fusionar. Esto es lo que se denomina “avance rápido” (“fast forward”).

Ahora, los cambios realizados están ya en la instantánea (snapshot) de la confirmación (commit) apuntada por la rama `master`. Y puedes desplegarlos.

**FIGURE 3-14**

Tras la fusión (merge), la rama `master` apunta al mismo sitio que la rama `hotfix`.

Tras haber resuelto el problema urgente que había interrumpido tu trabajo, puedes volver a donde estabas. Pero antes, es importante borrar la rama `hotfix`, ya que no la vamos a necesitar más, puesto que apunta exactamente al mismo sitio que la rama `master`. Esto lo puedes hacer con la opción `-d` del comando `git branch`:

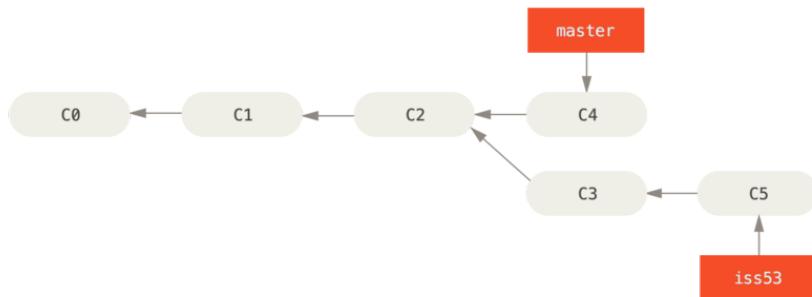
```
$ git branch -d hotfix
Deleted branch hotfix (3a0874c).
```

Y, con esto, ya estás listo para regresar al trabajo sobre el problema #53.

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

FIGURE 3-15

*La rama iss53
puede avanzar
independientemente*



Cabe destacar que todo el trabajo realizado en la rama hotfix no está en los archivos de la rama iss53. Si fuera necesario agregarlos, puedes fusionar (merge) la rama master sobre la rama iss53 utilizando el comando `git merge master`, o puedes esperar hasta que decidas fusionar (merge) la rama iss53 a la rama master.

Procedimientos Básicos de Fusión

Supongamos que tu trabajo con el problema #53 ya está completo y listo para fusionarlo (merge) con la rama master. Para ello, de forma similar a como antes has hecho con la rama hotfix, vas a fusionar la rama iss53. Simplemente, activa (checkout) la rama donde deseas fusionar y lanza el comando `git merge`:

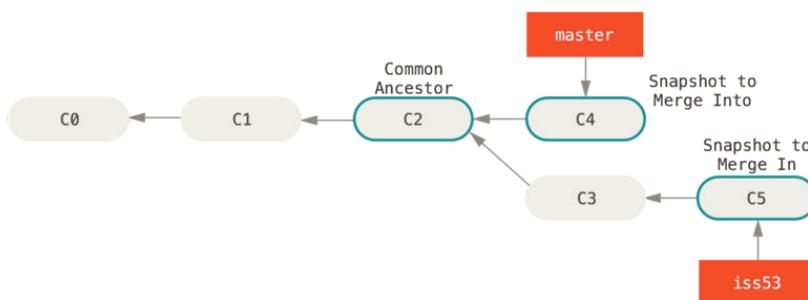
```
$ git checkout master
Switched to branch 'master'
$ git merge iss53
Merge made by the 'recursive' strategy.
```

```
index.html |    1 +
1 file changed, 1 insertion(+)
```

Es algo diferente de la fusión realizada anteriormente con hotfix. En este caso, el registro de desarrollo había divergido en un punto anterior. Debido a que la confirmación en la rama actual no es ancestro directo de la rama que pretendes fusionar, Git tiene cierto trabajo extra que hacer. Git realizará una fusión a tres bandas, utilizando las dos instantáneas apuntadas por el extremo de cada una de las ramas y por el ancestro común a ambas.

FIGURE 3-16

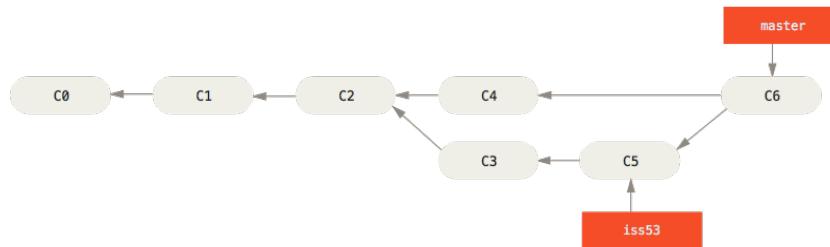
Git identifica automáticamente el mejor ancestro común para realizar la fusión de las ramas



En lugar de simplemente avanzar el apuntador de la rama, Git crea una nueva instantánea (snapshot) resultante de la fusión a tres bandas; y crea automáticamente una nueva confirmación de cambios (commit) que apunta a ella. Nos referimos a este proceso como “fusión confirmada” y su particularidad es que tiene más de un parent.

FIGURE 3-17

Git crea automáticamente una nueva confirmación para la fusión



Vale la pena destacar el hecho de que es el propio Git quien determina automáticamente el mejor ancestro común para realizar la fusión; a diferencia de otros sistemas tales como CVS o Subversion, donde es el desarrollador quien ha de determinar cuál puede ser dicho mejor ancestro común. Esto hace que en Git sea mucho más fácil realizar fusiones.

Ahora que todo tu trabajo ya está fusionado con la rama principal, no tienes necesidad de la rama `iss53`. Por lo que puedes borrarla y cerrar manualmente el problema en el sistema de seguimiento de problemas de tu empresa.

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

Principales Conflictos que Pueden Surgir en las Fusiones

En algunas ocasiones, los procesos de fusión no suelen ser fluidos. Si hay modificaciones dispares en una misma porción de un mismo archivo en las dos ramas distintas que pretendes fusionar, Git no será capaz de fusionarlas directamente. Por ejemplo, si en tu trabajo del problema #53 has modificado una misma porción que también ha sido modificada en el problema `hotfix`, verás un conflicto como este:

```
$ 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 no crea automáticamente una nueva fusión confirmada (merge commit), sino que hace una pausa en el proceso, esperando a que tú resuelvas el conflic-

to. Para ver qué archivos permanecen sin fusionar en un determinado momento conflictivo de una fusión, puedes usar el comando `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")
```

Todo aquello que sea conflictivo y no se haya podido resolver, se marca como “sin fusionar” (unmerged). Git añade a los archivos conflictivos unos marcadores especiales de resolución de conflictos que te guiarán cuando abras manualmente los archivos implicados y los edites para corregirlos. El archivo conflictivo contendrá algo como:

```
<<<<< 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
```

Donde nos dice que la versión en HEAD (la rama `master`, la que habías activado antes de lanzar el comando de fusión) contiene lo indicado en la parte superior del bloque (todo lo que está encima de `=====`) y que la versión en `iss53` contiene el resto, lo indicado en la parte inferior del bloque. Para resolver el conflicto, has de elegir manualmente el contenido de uno o de otro lado. Por ejemplo, puedes optar por cambiar el bloque, dejándolo así:

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

Esta corrección contiene un poco de ambas partes y se han eliminado completamente las líneas `<<<<<`, `=====` y `>>>>>`. Tras resolver todos los bloques conflictivos, has de lanzar comandos `git add` para marcar cada archi-

vo modificado. Marcar archivos como preparados (staged) indica a Git que sus conflictos han sido resueltos.

Si en lugar de resolver directamente prefieres utilizar una herramienta gráfica, puedes usar el comando `git mergetool`, el cual arrancará la correspondiente herramienta de visualización y te permitirá ir resolviendo conflictos con ella:

```
$ 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
Merging:
index.html

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

Si deseas usar una herramienta distinta de la escogida por defecto (en mi caso opendiff, porque estoy lanzando el comando en Mac), puedes escogerla entre la lista de herramientas soportadas mostradas al principio (“merge tool candidates”) tecleando el nombre de dicha herramienta.

Si necesitas herramientas más avanzadas para resolver conflictos de fusión más complicados, revisa la sección de fusiónado en “Advanced Merging”.

Tras salir de la herramienta de fusiónado, Git preguntará si hemos resuelto todos los conflictos y la fusión ha sido satisfactoria. Si le indicas que así ha sido, Git marca como preparado (staged) el archivo que acabamos de modificar. En cualquier momento, puedes lanzar el comando `git status` para ver si ya has resuelto todos los conflictos:

```
$ 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
```

Si todo ha ido correctamente, y ves que todos los archivos conflictivos están marcados como preparados, puedes lanzar el comando `git commit` para terminar de confirmar la fusión. El mensaje de confirmación por defecto será algo parecido a:

```
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
#
```

Puedes modificar este mensaje añadiendo detalles sobre cómo has resuelto la fusión, si lo consideras útil para que otros entiendan esta fusión en un futuro. Se trata de indicar por qué has hecho lo que has hecho; a no ser que resulte obvio, claro está.

Gestión de Ramas

Ahora que ya has creado, fusionado y borrado algunas ramas, vamos a dar un vistazo a algunas herramientas de gestión muy útiles cuando comienzas a utilizar ramas de manera avanzada.

El comando `git branch` tiene más funciones que las de crear y borrar ramas. Si lo lanzas sin parámetros, obtienes una lista de las ramas presentes en tu proyecto:

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

Fijate en el carácter * delante de la rama `master`: nos indica la rama activa en este momento (la rama a la que apunta HEAD). Si hacemos una confirmación de cambios (commit), esa será la rama que avance. Para ver la última confirmación de cambios en cada rama, puedes usar el comando `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 readmes
```

Otra opción útil para averiguar el estado de las ramas, es filtrarlas y mostrar solo aquellas que han sido fusionadas (o que no lo han sido) con la rama actualmente activa. Para ello, Git dispone de las opciones `--merged` y `--no-merged`. Si deseas ver las ramas que han sido fusionadas en la rama activa, puedes lanzar el comando `git branch --merged`:

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

Aparece la rama `iss53` porque ya ha sido fusionada. Las ramas que no llevan por delante el carácter * pueden ser eliminadas sin problemas, porque todo su contenido ya ha sido incorporado a otras ramas.

Para mostrar todas las ramas que contienen trabajos sin fusionar, puedes utilizar el comando `git branch --no-merged`:

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

Esto nos muestra la otra rama del proyecto. Debido a que contiene trabajos sin fusionar, al intentarla borrarla con `git branch -d`, el comando nos dará un error:

```
$ 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'.
```

Si realmente deseas borrar la rama, y perder el trabajo contenido en ella, puedes forzar el borrado con la opción `-D`; tal y como indica el mensaje de ayuda.

Flujos de Trabajo Ramificados

Ahora que ya has visto los procedimientos básicos de ramificación y fusión, ¿qué puedes o qué debes hacer con ellos? En este apartado vamos a ver algunos de los flujos de trabajo más comunes, de tal forma que puedas decidir si te gustaría incorporar alguno de ellos a tu ciclo de desarrollo.

Ramas de Largo Recorrido

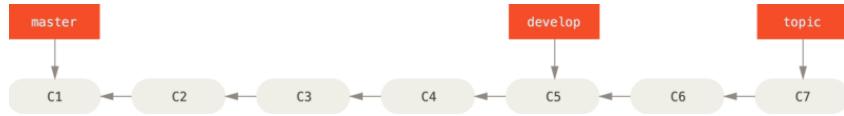
Por la sencillez de la fusión a tres bandas de Git, el fusionar una rama a otra varias veces a lo largo del tiempo es fácil de hacer. Esto te posibilita tener varias ramas siempre abiertas, eirlas usando en diferentes etapas del ciclo de desarrollo; realizando fusiones frecuentes entre ellas.

Muchos desarrolladores que usan Git llevan un flujo de trabajo de esta naturaleza, manteniendo en la rama `master` únicamente el código totalmente estable (el código que ha sido o que va a ser liberado) y teniendo otras ramas paralelas denominadas `desarrollo` o `siguiente`, en las que trabajan y realizan pruebas. Estas ramas paralelas no suele estar siempre en un estado estable; pero cada vez que sí lo están, pueden ser fusionadas con la rama `master`. También es habitual el incorporarle (`pull`) ramas puntuales (ramas temporales, como la rama `iss53` del ejemplo anterior) cuando las completamos y estamos seguros de que no van a introducir errores.

En realidad, en todo momento estamos hablando simplemente de apunadores moviéndose por la línea temporal de confirmaciones de cambio (`commit history`). Las ramas estables apuntan hacia posiciones más antiguas en el historial de confirmaciones, mientras que las ramas avanzadas, las que van abriendo camino, apuntan hacia posiciones más recientes.

FIGURE 3-18

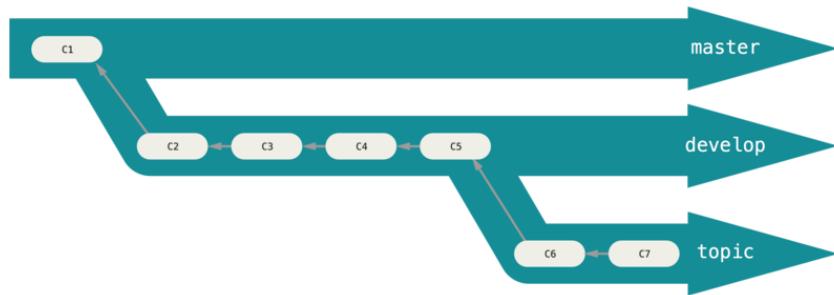
Una vista lineal del ramificado progresivo estable



Podría ser más sencillo pensar en las ramas como si fueran silos de almacenamiento, donde grupos de confirmaciones de cambio (commits) van siendo promocionados hacia silos más estables a medida que son probados y depurados.

FIGURE 3-19

Una vista tipo "silo" del ramificado progresivo estable



Este sistema de trabajo se puede ampliar para diversos grados de estabilidad. Algunos proyectos muy grandes suelen tener una rama denominada *prouestas* o *pu* (del inglés “proposed updates”, propuesta de actualización), donde suele estar todo aquello integrado desde otras ramas, pero que aún no está listo para ser incorporado a las ramas *siguiente* o *master*. La idea es mantener siempre diversas ramas en diversos grados de estabilidad; pero cuando alguna alcanza un estado más estable, la fusionamos con la rama inmediatamente superior a ella. Aunque no es obligatorio el trabajar con ramas de larga duración, realmente es práctico y útil, sobre todo en proyectos largos o complejos.

Ramas Puntuales

Las ramas puntuales, en cambio, son útiles en proyectos de cualquier tamaño. Una rama puntual es aquella rama de corta duración que abres para un tema o para una funcionalidad determinada. Es algo que nunca habrías hecho en otro sistema VCS, debido a los altos costos de crear y fusionar ramas en esos sistemas.

mas. Pero en Git, por el contrario, es muy habitual el crear, trabajar con, fusionar y eliminar ramas varias veces al día.

Tal y como has visto con las ramas `iss53` y `hotfix` que has creado en la sección anterior. Has hecho algunas confirmaciones de cambio en ellas, y luego las has borrado tras fusionarlas con la rama principal. Esta técnica te posibilita realizar cambios de contexto rápidos y completos y, debido a que el trabajo está claramente separado en silos, con todos los cambios de cada tema en su propia rama, te será mucho más sencillo revisar el código y seguir su evolución. Puedes mantener los cambios ahí durante minutos, días o meses; y fusionarlos cuando realmente estén listos, sin importar el orden en el que fueron creados o en el que comenzaste a trabajar en ellos.

Por ejemplo, puedes realizar cierto trabajo en la rama `master`, ramificar para un problema concreto (rama `iss91`), trabajar en él un rato, ramificar una segunda vez para probar otra manera de resolverlo (rama `iss92v2`), volver a la rama `master` y trabajar un poco más, y, por último, ramificar temporalmente para probar algo de lo que no estás seguro (rama `dumbidea`). El historial de confirmaciones (commit history) será algo parecido esto:

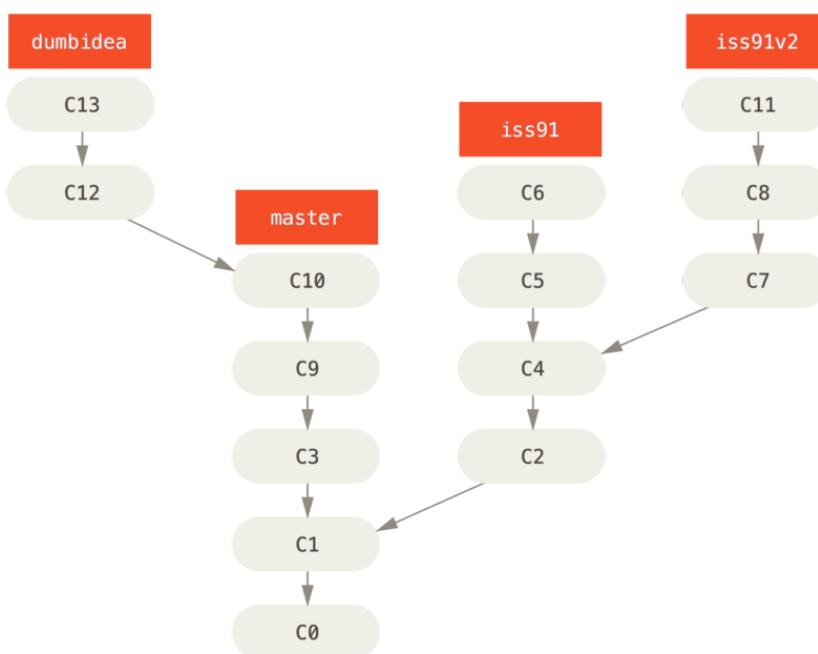
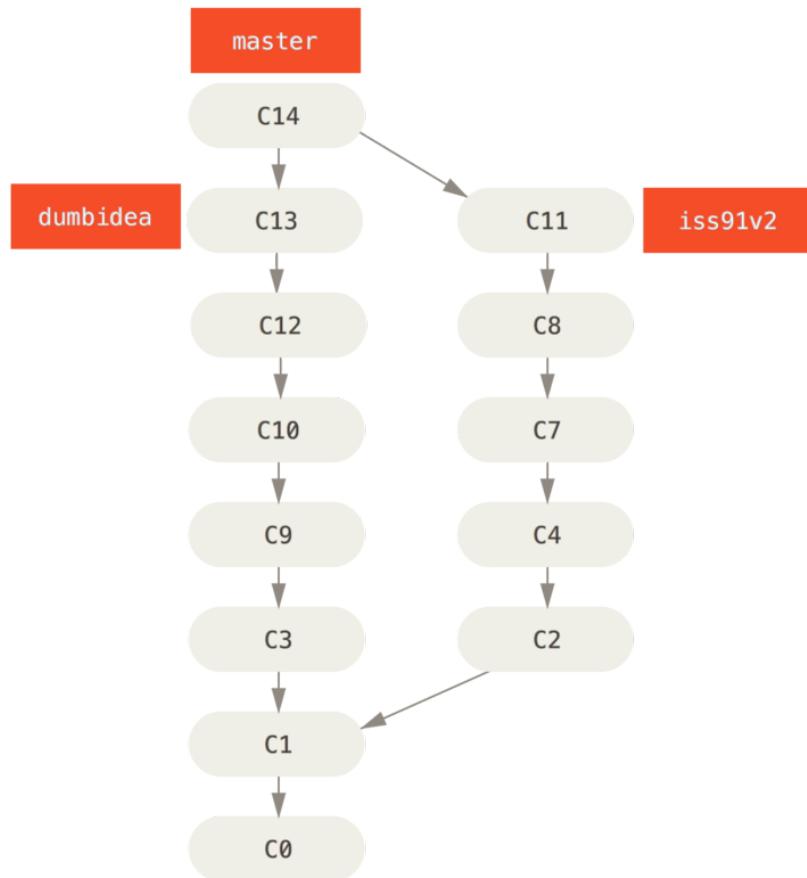


FIGURE 3-20
Múltiples ramas puntuales

En este momento, supongamos que te decides por la segunda solución al problema (rama `iss92v2`); y que, tras mostrar la rama `dumbidea` a tus compañeros, resulta que les parece una idea genial. Puedes descartar la rama `iss91` (perdiendo las confirmaciones C5 y C6), y fusionar las otras dos. El historial será algo parecido a esto:

FIGURE 3-21

El historial tras fusionar dumbidea e iss91v2



Hablaremos un poco más sobre los distintos flujos de trabajo de tu proyecto Git en **Chapter 5**, así que antes de decidir qué estilo de ramificación usará tu próximo proyecto, asegúrate de haber leído ese capítulo.

Es importante recordar que, mientras estás haciendo todo esto, todas las ramas son completamente locales. Cuando ramificas y fusionas, todo se realiza

en tu propio repositorio Git. No hay ningún tipo de comunicación con ningún servidor.

Ramas Remotas

Las ramas remotas son referencias al estado de las ramas en tus repositorios remotos. Son ramas locales que no puedes mover; se mueven automáticamente cuando estableces comunicaciones en la red. Las ramas remotas funcionan como marcadores, para recordarte en qué estado se encontraban tus repositorios remotos la última vez que conectaste con ellos.

Suelen referenciarse como `(remoto)/(rama)`. Por ejemplo, si quieres saber cómo estaba la rama `master` en el remoto `origin`, puedes revisar la rama `origin/master`. O si estás trabajando en un problema con un compañero y este envía (push) una rama `iss53`, tú tendrás tu propia rama de trabajo local `iss53`; pero la rama en el servidor apuntará a la última confirmación (commit) en la rama `origin/iss53`.

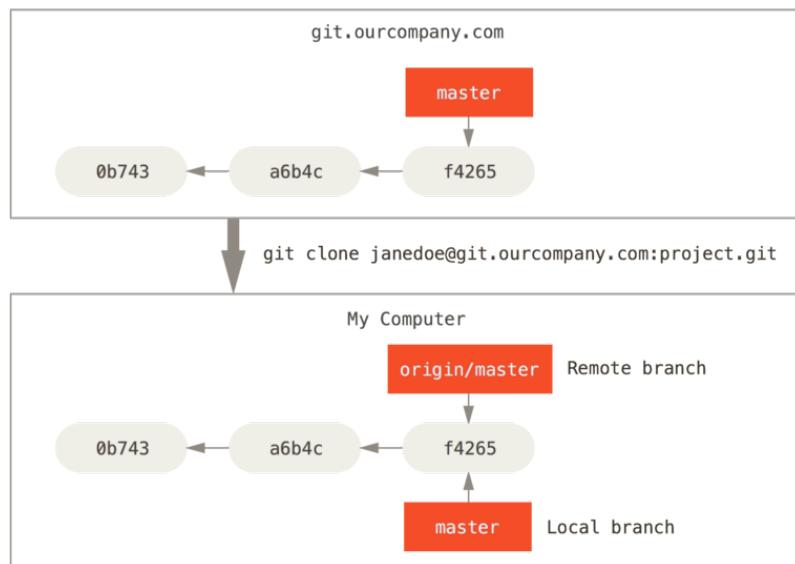
Esto puede ser un tanto confuso, pero intentemos aclararlo con un ejemplo. Supongamos que tienes un servidor Git en tu red, en `git.ourcompany.com`. Si haces un clón desde ahí, Git automáticamente lo denominará `origin`, traerá (pull) sus datos, creará un apuntador hacia donde esté en ese momento su rama `master` y denominará la copia local `origin/master`. Git te proporcionará también tu propia rama `master`, apuntando al mismo lugar que la rama `master` de `origin`; de manera que tengas donde trabajar.

EXAMPLE 3-1. “origin” no es especial

Así como la rama “`master`” no tiene ningún significado especial en Git, tampoco lo tiene “`origin`”. “`master`” es un nombre muy usado solo porque es el nombre por defecto que Git le da a la rama inicial cuando ejecutas `git init`. De la misma manera, “`origin`” es el nombre por defecto que Git le da a un remoto cuando ejecutas `git clone`. Si en cambio ejecutases `git clone -o booyah`, tendrías una rama `booyah/master` como rama remota por defecto.

FIGURE 3-22

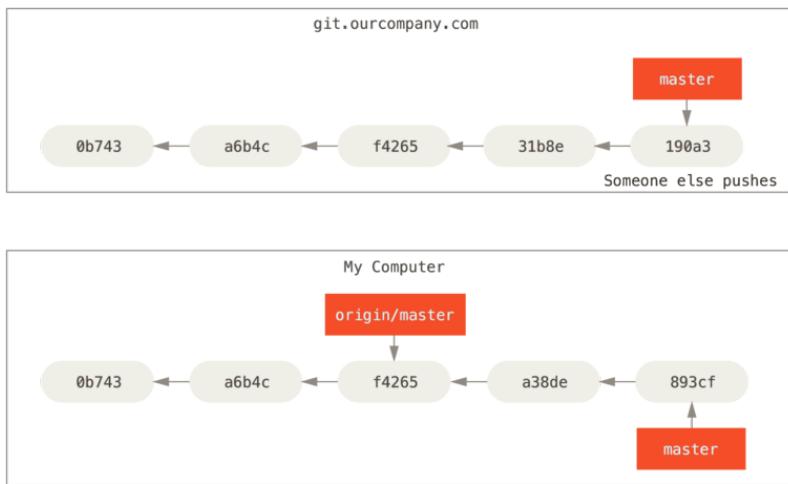
Servidor y
repositorio local
luego de ser clonado



Si haces algún trabajo en tu rama `master` local, y al mismo tiempo, alguien más lleva (push) su trabajo al servidor `git.ourcompany.com`, actualizando la rama `master` de allí, te encontrarás con que ambos registros avanzan de forma diferente. Además, mientras no tengas contacto con el servidor, tu apuntador a tu rama `origin/master` no se moverá.

FIGURE 3-23

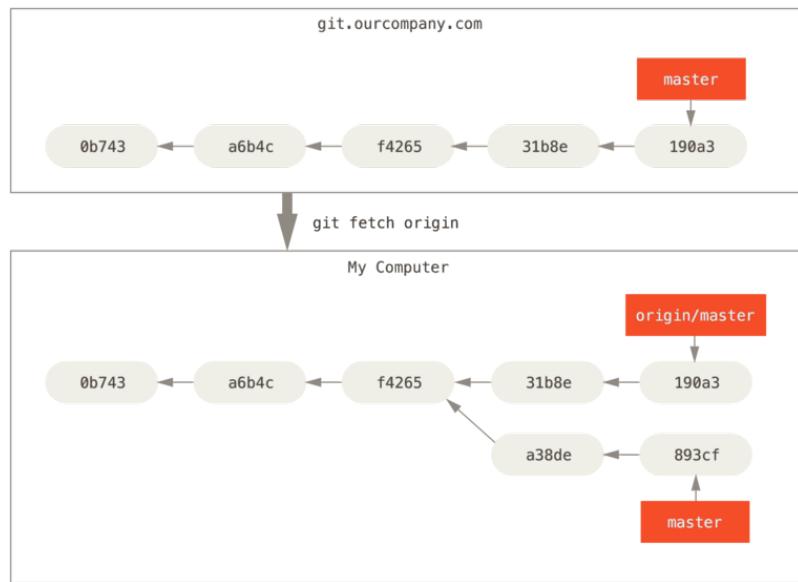
El trabajo remoto y el local pueden diverger



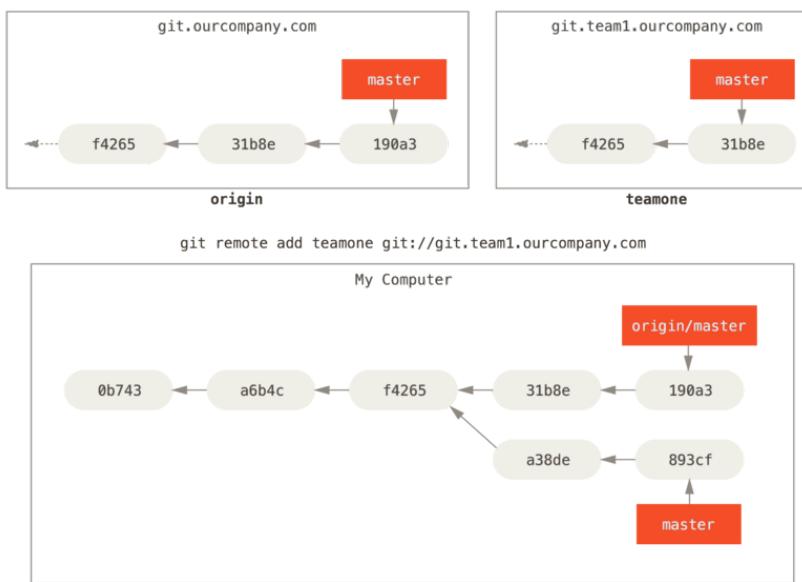
Para sincronizarte, puedes utilizar el comando `git fetch origin`. Este comando localiza en qué servidor está el origen (en este caso `git.ourcompany.com`), recupera cualquier dato presente allí que tú no tengas, y actualiza tu base de datos local, moviendo tu rama `origin/master` para que apunte a la posición más reciente.

FIGURE 3-24

git fetch actualiza las referencias de tu remoto



Para ilustrar mejor el caso de tener múltiples servidores y cómo van las ramas remotas para esos proyectos remotos, supongamos que tienes otro servidor Git; utilizado por uno de tus equipos sprint, solamente para desarrollo. Este servidor se encuentra en `git.team1.ourcompany.com`. Puedes incluirlo como una nueva referencia remota a tu proyecto actual, mediante el comando `git remote add`, tal y como vimos en [Chapter 2](#). Puedes denominar `teamone` a este remoto al asignarle este nombre a la URL.

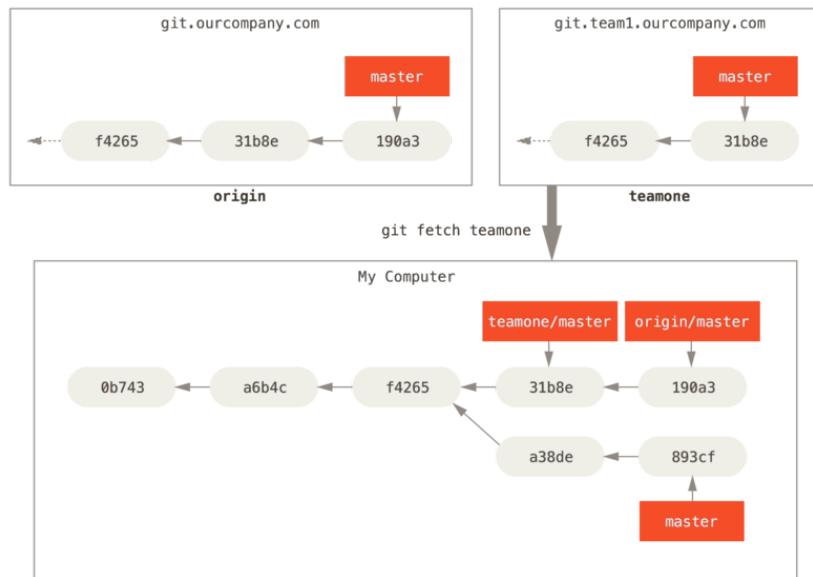
**FIGURE 3-25**

Añadiendo otro servidor como remoto

Ahora, puedes usar el comando `git fetch teamone` para recuperar todo el contenido del remoto `teamone` que tú no tenías. Debido a que dicho servidor es un subconjunto de los datos del servidor `origin` que tienes actualmente, Git no recupera (fetch) ningún dato; simplemente prepara una rama remota llamada `teamone/master` para apuntar a la confirmación (commit) que `teamone` tiene en su rama `master`.

FIGURE 3-26

*Seguimiento de la
rama remota a
través de teamone/
master*



Publicar

Cuando quieres compartir una rama con el resto del mundo, debes llevarla (push) a un remoto donde tengas permisos de escritura. Tus ramas locales no se sincronizan automáticamente con los remotos en los que escribes, sino que tienes que enviar (push) expresamente las ramas que deseas compartir. De esta forma, puedes usar ramas privadas para el trabajo que no deseas compartir, llevando a un remoto tan solo aquellas partes que deseas aportar a los demás.

Si tienes una rama llamada `serverfix`, con la que vas a trabajar en colaboración; puedes llevarla al remoto de la misma forma que llevaste tu primera rama. Con el comando `git push (remoto) (rama)`:

```
$ 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
```

Esto es un atajo. Git expande automáticamente el nombre de rama `serverfix` a `refs/heads/serverfix:refs/heads/serverfix`, que significa: “coge mi rama local `serverfix` y actualiza con ella la rama `serverfix` del remoto”. Volveremos más tarde sobre el tema de `refs/heads/`, viéndolo en detalle en **Chapter 10**; por ahora, puedes ignorarlo. También puedes hacer `git push origin serverfix:serverfix`, que hace lo mismo; es decir: “coge mi `serverfix` y hazlo el `serverfix` remoto”. Puedes utilizar este último formato para llevar una rama local a una rama remota con un nombre distinto. Si no quieres que se llame `serverfix` en el remoto, puedes lanzar, por ejemplo, `git push origin serverfix:awesomebranch`; para llevar tu rama `serverfix` local a la rama `awesomebranch` en el proyecto remoto.

NO ESCRIBAS TU CONTRASEÑA TODO EL TIEMPO

Si utilizas una dirección URL con HTTPS para enviar datos, el servidor Git te preguntará tu usuario y contraseña para autenticarte. Por defecto, te pedirá esta información a través del terminal, para determinar si estás autorizado a enviar datos.

Si no quieres escribir tu contraseña cada vez que haces un envío, puedes establecer un “cache de credenciales”. La manera más sencilla de hacerlo es estableciéndolo en memoria por unos minutos, lo que puedes lograr fácilmente al ejecutar `git config --global credential.helper cache`

Para más información sobre las distintas opciones de cache de credenciales, véase “[Credential Storage](#)”.

La próxima vez que tus colaboradores recuperen desde el servidor, obtendrán bajo la rama remota `origin/serverfix` una referencia a donde esté la versión de `serverfix` en el servidor:

```
$ 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
```

Es importante destacar que cuando recuperas (fetch) nuevas ramas remotas, no obtienes automáticamente una copia local editable de las mismas. En otras palabras, en este caso, no tienes una nueva rama `serverfix`. Sino que únicamente tienes un puntero no editable a `origin/serverfix`.

Para integrar (merge) esto en tu rama de trabajo actual, puedes usar el comando `git merge origin/serverfix`. Y si quieres tener tu propia rama `serverfix` para trabajar, puedes crearla directamente basandote en la rama remota:

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

Esto sí te da una rama local donde puedes trabajar, que comienza donde `origin/serverfix` estaba en ese momento.

Hacer Seguimiento a las Ramas

Al activar (checkout) una rama local a partir de una rama remota, se crea automáticamente lo que podríamos denominar una “rama de seguimiento” (tracking branch). Las ramas de seguimiento son ramas locales que tienen una relación directa con alguna rama remota. Si estás en una rama de seguimiento y tecleas el comando `git pull`, Git sabe de cuál servidor recuperar (fetch) y fusionar (merge) datos.

Cuando clonas un repositorio, este suele crear automáticamente una rama `master` que hace seguimiento de `origin/master`. Sin embargo, puedes preparar otras ramas de seguimiento si deseas tener unas que sigan ramas de otros remotos o no seguir la rama `master`. El ejemplo más simple es el que acabas de ver al lanzar el comando `git checkout -b [rama] [nombreremoto]/[rama]`. Esta operación es tan común que git ofrece el parámetro `--track`:

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

Para preparar una rama local con un nombre distinto a la del remoto, puedes utilizar la primera versión con un nombre de rama local diferente:

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

Así, tu rama local `sf` traerá (pull) información automáticamente desde `origin/serverfix`.

Si ya tienes una rama local y quieres asignarla a una rama remota que acabas de traerte, o quieres cambiar la rama a la que le haces seguimiento, puedes usar en cualquier momento las opciones `-u` o `--set-upstream-to` del comando `git branch`.

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

ATAJO AL UPSTREAM

Cuando tienes asignada una rama de seguimiento, puedes hacer referencia a ella mediante `@{upstream}` o mediante el atajo `@{u}`. De esta manera, si estás en la rama `master` y esta sigue a la rama `origin/master`, puedes hacer algo como `git merge @{u}` en vez de `git merge origin/master`.

Si quieres ver las ramas de seguimiento que tienes asignado, puedes usar la opción `-vv` con `git branch`. Esto listará tus ramas locales con más información, incluyendo a qué sigue cada rama y si tu rama local está por delante, por detrás o ambas.

```
$ git branch -vv
iss53      7e424c3 [origin/iss53: ahead 2] forgot the brackets
master     1ae2a45 [origin/master] deploying index fix
* serverfix f8674d9 [teamone/server-fix-good: ahead 3, behind 1] this should do it
  testing   5ea463a trying something new
```

Aquí podemos ver que nuestra rama `iss53` sigue `origin/iss53` y está “ahead” (delante) por dos, es decir, que tenemos dos confirmaciones locales que no han sido enviadas al servidor. También podemos ver que nuestra rama `master` sigue a `origin/master` y está actualizada. Luego podemos ver que nuestra rama `serverfix` sigue la rama `server-fix-good` de nuestro servidor `teamone` y que está tres cambios por delante (ahead) y uno por detrás (behind), lo que significa que existe una confirmación en el servidor que no hemos fusionado y que tenemos tres confirmaciones locales que no hemos enviado. Por último, podemos ver que nuestra rama `testing` no sigue a ninguna rama remota.

Es importante destacar que estos números se refieren a la última vez que trajiste (fetch) datos de cada servidor. Este comando no se comunica con los servidores, solo te indica lo que sabe de ellos localmente. Si quieres tener los

cambios por delante y por detrás actualizados, debes traerlos (fetch) de cada servidor antes de ejecutar el comando. Puedes hacerlo de esta manera: `$ git fetch --all; git branch -vv`

Traer y Fusionar

A pesar de que el comando `git fetch` trae todos los cambios del servidor que no tienes, este no modifica tu directorio de trabajo. Simplemente obtendrá los datos y dejará que tú mismo los fusiones. Sin embargo, existe un comando llamado `git pull`, el cuál básicamente hace `git fetch` seguido por `git merge` en la mayoría de los casos. Si tienes una rama de seguimiento configurada como vimos en la última sección, bien sea asignándola explícitamente o creándola mediante los comandos `clone` o `checkout`, `git pull` identificará a qué servidor y rama remota sigue tu rama actual, traerá los datos de dicho servidor e intentará fusionar dicha rama remota.

Normalmente es mejor usar los comandos `fetch` y `merge` de manera explícita pues la magia de `git pull` puede resultar confusa.

Eliminar Ramas Remotas

Imagina que ya has terminado con una rama remota, es decir, tanto tú como tus colaboradores habéis completado una determinada funcionalidad y la habéis incorporado (merge) a la rama `master` en el remoto (o donde quiera que tengáis la rama de código estable). Puedes borrar la rama remota utilizando la opción `--delete` de `git push`. Por ejemplo, si quieres borrar la rama `serverfix` del servidor, puedes utilizar:

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

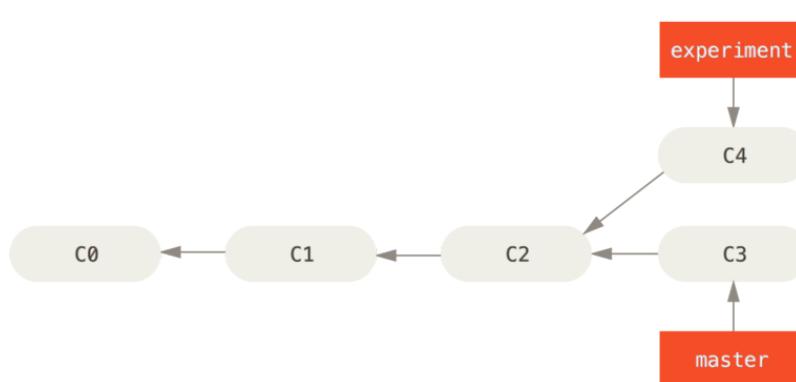
Básicamente lo que hace es eliminar el apuntador del servidor. El servidor Git suele mantener los datos por un tiempo hasta que el recolector de basura se ejecute, de manera que si la has borrado accidentalmente, suele ser fácil recuperarla.

Reorganizar el Trabajo Realizado

En Git tenemos dos formas de integrar cambios de una rama en otra: la fusión (merge) y la reorganización (rebase). En esta sección vas a aprender en qué consiste la reorganización, cómo utilizarla, por qué es una herramienta sorprendente y en qué casos no es conveniente utilizarla.

Reorganización Básica

Volviendo al ejemplo anterior, en la sección sobre fusiones “**Procedimientos Básicos de Fusión**” puedes ver que has separado tu trabajo y realizado confirmaciones (commit) en dos ramas diferentes.

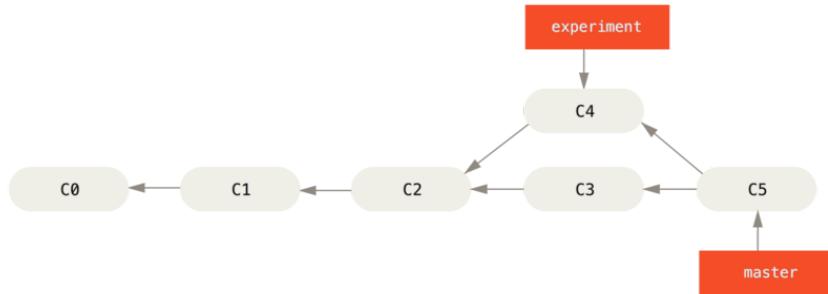
**FIGURE 3-27**

El registro de confirmaciones inicial

La manera más sencilla de integrar ramas, tal y como hemos visto, es el comando `git merge`. Realiza una fusión a tres bandas entre las dos últimas instantáneas de cada rama (C3 y C4) y el ancestro común a ambas (C2); creando una nueva instantánea (snapshot) y la correspondiente confirmación (commit).

FIGURE 3-28

Fusionar una rama para integrar el registro de trabajos divergentes



Sin embargo, también hay otra forma de hacerlo: puedes coger los cambios introducidos en C3 y reaplicarlos encima de C4. Esto es lo que en Git llamamos *reorganizar* (*rebasing*, en inglés). Con el comando `git rebase`, puedes coger todos los cambios confirmados en una rama, y reaplicarlos sobre otra.

Por ejemplo, puedes lanzar los comandos:

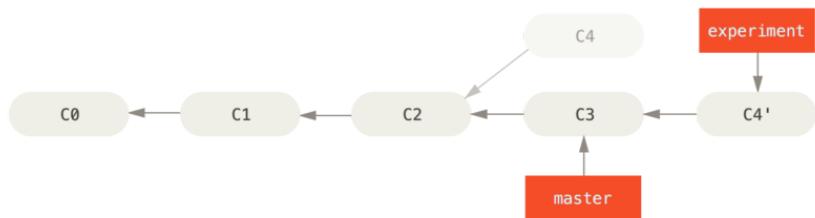
```

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

Haciendo que Git vaya al ancestro común de ambas ramas (donde estás actualmente y de donde quieres reorganizar), saque las diferencias introducidas por cada confirmación en la rama donde estás, guarde esas diferencias en archivos temporales, reinicie (reset) la rama actual hasta llevarla a la misma confirmación en la rama de donde quieres reorganizar, y, finalmente, vuelva a aplicar ordenadamente los cambios.

FIGURE 3-29

Reorganizando sobre C3 los cambios introducidos en C4



En este momento, puedes volver a la rama `master` y hacer una fusión con avance rápido (fast-forward merge).

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

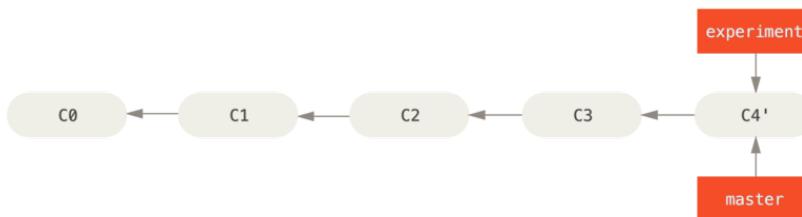


FIGURE 3-30
Avance rápido de la
rama `master`

Así, la instantánea apuntada por `C4'` es exactamente la misma apuntada por `C5` en el ejemplo de la fusión. No hay ninguna diferencia en el resultado final de la integración, pero el haberla hecho reorganizando nos deja un historial más claro. Si examinas el historial de una rama reorganizada, este aparece siempre como un historial lineal: como si todo el trabajo se hubiera realizado en series, aunque realmente se haya hecho en paralelo.

Habitualmente, optarás por esta vía cuando quieras estar seguro de que tus confirmaciones de cambio (commits) se pueden aplicar limpiamente sobre una rama remota; posiblemente, en un proyecto donde estés intentando colaborar, pero lleves tú el mantenimiento. En casos como esos, puedes trabajar sobre una rama y luego reorganizar lo realizado en la rama `origin/master` cuando lo tengas todo listo para enviarlo al proyecto principal. De esta forma, la persona que mantiene el proyecto no necesitará hacer ninguna integración con tu trabajo; le bastará con un avance rápido o una incorporación limpia.

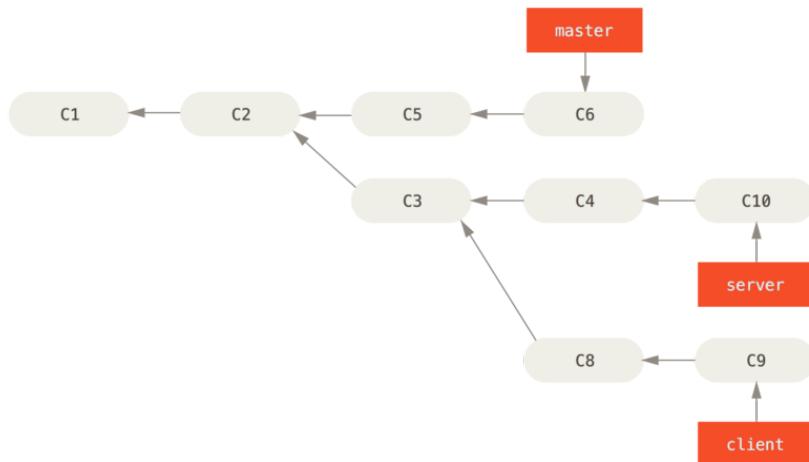
Cabe destacar que la instantánea (snapshot) apuntada por la confirmación (commit) final, tanto si es producto de una reorganización (rebase) como si lo es de una fusión (merge), es exactamente la misma instantánea; lo único diferente es el historial. La reorganización vuelve a aplicar cambios de una rama de trabajo sobre otra rama, en el mismo orden en que fueron introducidos en la primera, mientras que la fusión combina entre sí los dos puntos finales de ambas ramas.

Algunas Reorganizaciones Interesantes

También puedes aplicar una reorganización (rebase) sobre otra cosa además de sobre la rama de reorganización. Por ejemplo, considera un historial como el de **Figure 3-31**. Has ramificado a una rama puntual (`server`) para añadir algunas funcionalidades al proyecto, y luego has confirmado los cambios. Después, vuelves a la rama original para hacer algunos cambios en la parte cliente (rama `client`), y confirmas también esos cambios. Por último, vuelves sobre la rama `server` y haces algunos cambios más.

FIGURE 3-31

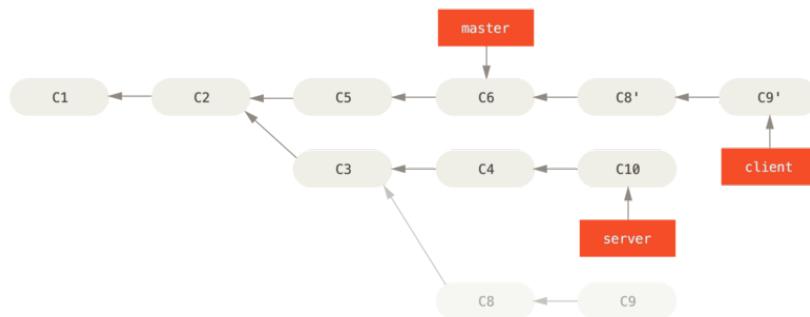
Un historial con una rama puntual sobre otra rama puntual



Imagina que decides incorporar tus cambios del lado cliente sobre el proyecto principal para hacer un lanzamiento de versión; pero no quieres lanzar aún los cambios del lado servidor porque no están aún suficientemente probados. Puedes coger los cambios del cliente que no están en server (C8 y C9) y reaplicarlos sobre tu rama principal usando la opción `--onto` del comando `git rebase`:

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

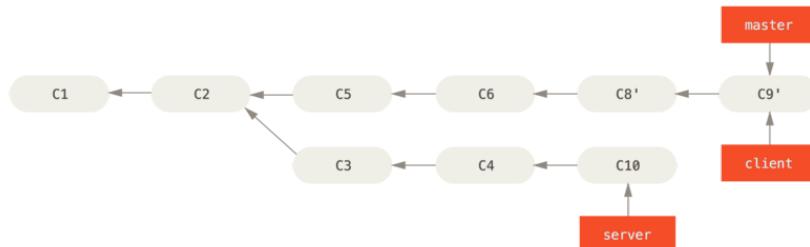
Esto viene a decir: “Activa la rama `client`, averigua los cambios desde el ancestro común entre las ramas `client` y `server`, y aplícalos en la rama `master`”. Puede parecer un poco complicado, pero los resultados son realmente interesantes.

**FIGURE 3-32**

Reorganizando una rama puntual fuera de otra rama puntual

Y, tras esto, ya puedes avanzar la rama principal (ver **Figure 3-33**):

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

**FIGURE 3-33**

Avance rápido de tu rama master, para incluir los cambios de la rama client

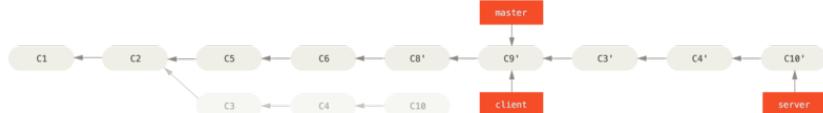
Ahora supongamos que decides traerlos (pull) también sobre tu rama **server**. Puedes reorganizar (rebase) la rama **server** sobre la rama **master** sin necesidad siquiera de comprobarlo previamente, usando el comando `git rebase [rama-base] [rama-puntual]`, el cual activa la rama puntual (**server** en este caso) y la aplica sobre la rama base (**master** en este caso):

```
$ git rebase master server
```

Esto vuelca el trabajo de **server** sobre el de **master**, tal y como se muestra en **Figure 3-34**.

FIGURE 3-34

Reorganizando la rama server sobre la rama master



Después, puedes avanzar rápidamente la rama base (`master`):

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

Y por último puedes eliminar las ramas `client` y `server` porque ya todo su contenido ha sido integrado y no las vas a necesitar más, dejando tu registro tras todo este proceso tal y como se muestra en **Figure 3-35**:

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

FIGURE 3-35

Historial final de confirmaciones de cambio



Los Peligros de Reorganizar

Ahh..., pero la dicha de la reorganización no la alcanzamos sin sus contrapartidas, las cuales pueden resumirse en una línea:

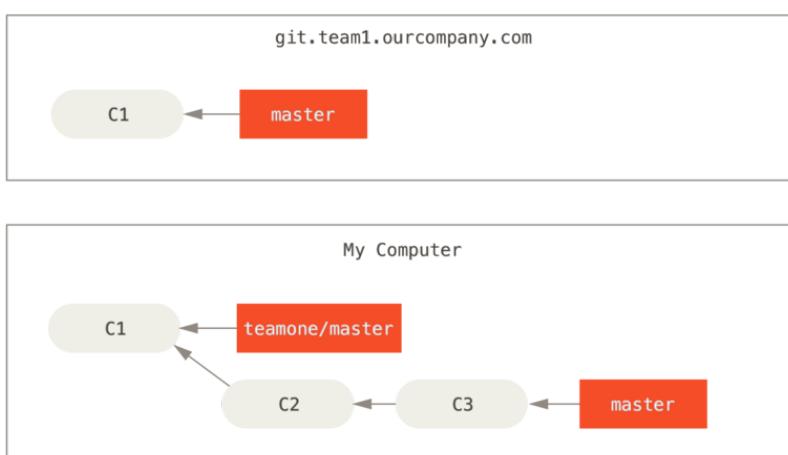
Nunca reorganices confirmaciones de cambio (commits) que hayas enviado (push) a un repositorio público.

Si sigues esta recomendación, no tendrás problemas. Pero si no lo haces, la gente te odiará y serás despreciado por tus familiares y amigos.

Cuando reorganizas algo, estás abandonando las confirmaciones de cambio ya creadas y estás creando unas nuevas; que son similares, pero diferentes. Si envías (push) confirmaciones (commits) a alguna parte, y otros las recogen (pull) de allí; y después vas tú y las reescribes con `git rebase` y las vuelves a enviar (push); tus colaboradores tendrán que refusionar (re-merge) su trabajo y

todo se volverá tremadamente complicado cuando intentes recoger (pull) su trabajo de vuelta sobre el tuyo.

Veamos con un ejemplo como reorganizar trabajo que has hecho público puede causar problemas. Imagínate que haces un clon desde un servidor central, y luego trabajas sobre él. Tu historial de cambios puede ser algo como esto:

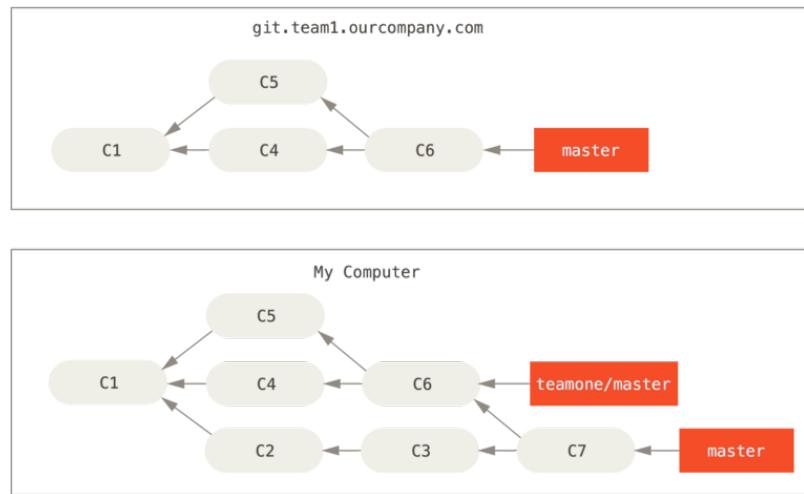
**FIGURE 3-36**

Clonar un repositorio y trabajar sobre él

Ahora, otra persona trabaja también sobre ello, realiza una fusión (merge) y lleva (push) su trabajo al servidor central. Tú te traes (fetch) sus trabajos y los fusionas (merge) sobre una nueva rama en tu trabajo, con lo que tu historial quedaría parecido a esto:

FIGURE 3-37

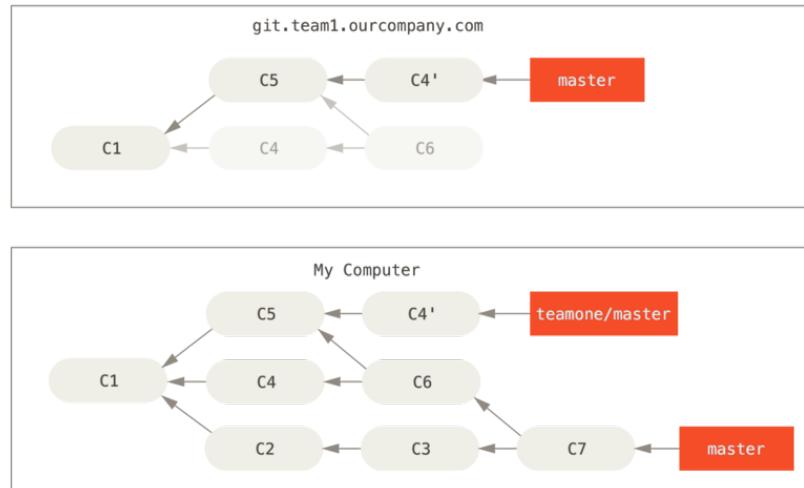
Traer (fetch) algunas confirmaciones de cambio (commits) y fusionarlas (merge) sobre tu trabajo



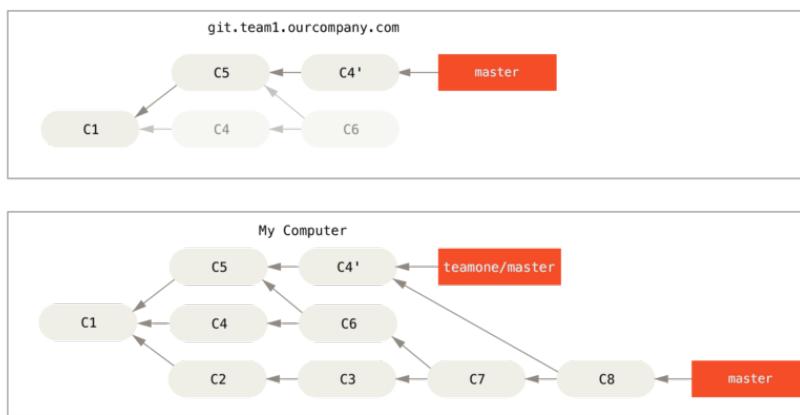
A continuación, la persona que había llevado cambios al servidor central decide retroceder y reorganizar su trabajo; haciendo un `git push --force` para sobrescribir el registro en el servidor. Tu te traes (fetch) esos nuevos cambios desde el servidor.

FIGURE 3-38

Alguien envio (push) confirmaciones (commits) reorganizadas, abandonando las confirmaciones en las que tu habías basado tu trabajo



Ahora los dos están en un aprieto. Si haces `git pull` crearás una fusión confirmada, la cual incluirá ambas líneas del historial, y tu repositorio lucirá así:

**FIGURE 3-39**

Vuelves a fusionar el mismo trabajo en una nueva fusión confirmada

Si ejecutas `git log` sobre un historial así, verás dos confirmaciones hechas por el mismo autor y con la misma fecha y mensaje, lo cual será confuso. Es más, si luego tu envías (`push`) ese registro de vuelta al servidor, vas a introducir todas esas confirmaciones reorganizadas en el servidor central. Lo que puede confundir aún más a la gente. Era más seguro asumir que el otro desarrollador no quería que C4 y C6 estuviesen en el historial; por ello había reorganizado su trabajo de esa manera.

Reorganizar una Reorganización

Si te encuentras en una situación como esta, Git tiene algunos trucos que pueden ayudarte. Si alguien de tu equipo sobreescribe cambios en los que se basaba tu trabajo, tu reto es descubrir qué han sobreescrito y qué te pertenece.

Además de la suma de control SHA-1, Git calcula una suma de control basada en el parche que introduce una confirmación. A esta se le conoce como "patch-id".

Si te traes el trabajo que ha sido sobreescrito y lo reorganizas sobre las nuevas confirmaciones de tu compañero, es posible que Git pueda identificar qué parte correspondía específicamente a tu trabajo y aplicarla de vuelta en la rama nueva.

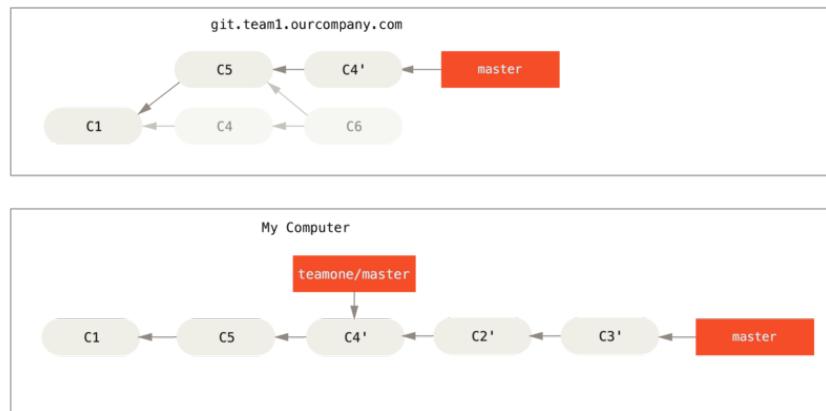
Por ejemplo, en el caso anterior, si en vez de hacer una fusión cuando estábamos en **Figure 3-38** ejecutamos `git rebase teamone/master`, Git hará lo siguiente:

- Determinar el trabajo que es específico de nuestra rama (C2, C3, C4, C6, C7)
- Determinar cuáles no son fusiones confirmadas (C2, C3, C4)
- Determinar cuáles no han sido sobreescritas en la rama destino (solo C2 y C3, pues C4 corresponde al mismo parche que C4')
- Aplicar dichas confirmaciones encima de `teamone/master`

Así que en vez del resultado que vimos en **Figure 3-39**, terminaremos con algo más parecido a **Figure 3-40**.

FIGURE 3-40

Reorganizar encima de un trabajo sobreescrito reorganizado.



Esto solo funciona si C4 y el C4' de tu compañero son parches muy similares. De lo contrario, la reorganización no será capaz de identificar que se trata de un duplicado y agregaría otro parche similar a C4 (lo cual probablemente no podrá aplicarse limpiamente, pues los cambios ya estarían allí en algún lugar).

También puedes simplificar el proceso si ejecutas `git pull --rebase` en vez del tradicional `git pull`. O, en este caso, puedes hacerlo manualmente con un `git fetch` primero, seguido de un `git rebase teamone/master`.

Si sueles utilizar `git pull` y quieres que la opción `--rebase` esté activada por defecto, puedes asignar el valor de configuración `pull.rebase` haciendo algo como esto `git config --global pull.rebase true`.

Si consideras la reorganización como una manera de limpiar tu trabajo y tus confirmaciones antes de enviarlas (push), y si solo reorganizas confirmaciones (commits) que nunca han estado disponibles públicamente, no tendrás problemas. Si reorganizas (rebase) confirmaciones (commits) que ya estaban disponibles públicamente y la gente había basado su trabajo en ellas, entonces prepárate para tener problemas, frustrar a tu equipo y ser despreciado por tus compañeros.

Si tu compañero o tú ven que aun así es necesario hacerlo en algún momento, asegúrense que todos sepan que deben ejecutar `git pull --rebase` para intentar aliviar en lo posible la frustración.

Reorganizar vs. Fusionar

Ahora que has visto en acción la reorganización y la fusión, te preguntarás cuál es mejor. Antes de responder, repasemos un poco qué representa el historial.

Para algunos, el historial de confirmaciones de tu repositorio es **un registro de todo lo que ha pasado**. Un documento histórico, valioso por sí mismo y que no debería ser alterado. Desde este punto de vista, cambiar el historial de confirmaciones es casi como blasfemar; estarías *mintiendo* sobre lo que en verdad ocurrió. ¿Y qué pasa si hay una serie desastrosa de fusiones confirmadas? Nada. Así fue como ocurrió y el repositorio debería tener un registro de esto para la posteridad.

La otra forma de verlo es que el historial de confirmaciones es **la historia de cómo se hizo tu proyecto**. Tú no publicarías el primer borrador de tu novela, y el manual de cómo mantener tus programas también debe estar editado con mucho cuidado. Esta es el área que utiliza herramientas como `rebase` y `filter-branch` para contar la historia de la mejor manera para los futuros lectores.

Ahora, sobre qué es mejor si fusionar o reorganizar: verás que la respuesta no es tan sencilla. Git es una herramienta poderosa que te permite hacer muchas cosas con tu historial, y cada equipo y cada proyecto es diferente. Ahora que conoces cómo trabajan ambas herramientas, será cosa tuya decidir cuál de las dos es mejor para tu situación en particular.

Normalmente, la manera de sacar lo mejor de ambas es reorganizar tu trabajo local, que aun no has compartido, antes de enviarlo a algún lugar; pero nunca reorganizar nada que ya haya sido compartido.

Recapitulación

Hemos visto los procedimientos básicos de ramificación (branching) y fusión (merging) en Git. A estas alturas, te sentirás cómodo creando nuevas ramas

(branch), saltando (checkout) entre ramas para trabajar y fusionando (merge) ramas entre ellas. También conocerás cómo compartir tus ramas enviándolas (push) a un servidor compartido, cómo trabajar colaborativamente en ramas compartidas, y cómo reorganizar (rebase) tus ramas antes de compartirlas. A continuación, hablaremos sobre lo que necesitas para tener tu propio servidor de hospedaje Git.

Git en el Servidor

4

En este punto, deberías ser capaz de realizar la mayoría de las tareas diarias para las cuales estarás usando Git. Sin embargo, para poder realizar cualquier colaboración en Git, necesitarás tener un repositorio remoto Git. Aunque técnicamente puedes enviar y recibir cambios desde repositorios de otros individuos, no se recomienda hacerlo porque, si no tienes cuidado, fácilmente podrías confundir en que es en lo que se está trabajando. Además, lo deseable es que tus colaboradores sean capaces de acceder al repositorio incluso si tu computadora no está en línea – muchas veces es útil tener un repositorio confiable en común. Por lo tanto, el método preferido para colaborar con otra persona es configurar un repositorio intermedio al cual ambos tengan acceso, y enviar (push) y recibir (pull) desde allí.

Poner en funcionamiento un servidor Git es un proceso bastante claro. Primero, eliges con qué protocolos ha de comunicarse tu servidor. La primera sección de este capítulo cubrirá los protocolos disponibles, así como los pros y los contras de cada uno. Las siguientes secciones explicarán algunas configuraciones comunes utilizando dichos protocolos y como poner a funcionar tu servidor con alguno de ellos. Finalmente, revisaremos algunas de las opciones hospedadas, si no te importa hospedar tu código en el servidor de alguien más y no quieres tomarte la molestia de configurar y mantener tu propio servidor.

Si no tienes interés en tener tu propio servidor, puedes saltarte hasta la última sección de este capítulo para ver algunas de las opciones para configurar una cuenta hospedada y seguir al siguiente capítulo, donde discutiremos los varios pormenores de trabajar en un ambiente de control de fuente distribuído.

Un repositorio remoto es generalmente un *repositorio básico* – un repositorio Git que no tiene directorio de trabajo. Dado que el repositorio es solamente utilizado como un punto de colaboración, no hay razón para tener una copia instantánea verificada en el disco; tan solo son datos Git. En los más simples términos, un repositorio básico es el contenido `.git` del directorio de tu proyecto y nada más.

Los Protocolos

Git puede usar cuatro protocolos principales para transferir datos: Local, HTTP, Secure Shell (SSH) y Git. Vamos a ver en qué consisten y las circunstancias en que querrás (o no) utilizar cada uno de ellos.

Local Protocol

El más básico es el *Protocolo Local*, donde el repositorio remoto es simplemente otra carpeta en el disco. Se utiliza habitualmente cuando todos los miembros del equipo tienen acceso a un mismo sistema de archivos, como por ejemplo un punto de montaje NFS, o en el caso menos frecuente de que todos se conectan al mismo ordenador. Aunque este último caso no es precisamente el ideal, ya que todas las instancias del repositorio estarían en la misma máquina; aumentando las posibilidades de una pérdida catastrófica.

Si dispones de un sistema de archivos compartido, podrás clonar (clone), enviar (push) y recibir (pull) a/desde repositorios locales basado en archivos. Para clonar un repositorio como estos, o para añadirlo como remoto a un proyecto ya existente, usa el camino (path) del repositorio como su URL. Por ejemplo, para clonar un repositorio local, puedes usar algo como:

```
$ git clone /opt/git/project.git
```

O como:

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

Git trabaja ligeramente distinto si indicas *file://* de forma explícita al comienzo de la URL. Si escribes simplemente el camino, Git intentará usar enlaces rígidos (hardlinks) o copiar directamente los archivos que necesita. Si escribes con el prefijo *file://*, Git lanza el proceso que usa habitualmente para transferir datos sobre una red; proceso que suele ser mucho menos eficiente. La única razón que puedes tener para indicar expresamente el prefijo *file://* puede ser el querer una copia limpia del repositorio, descartando referencias u objetos superfluos. Esto sucede normalmente, tras haberlo importado desde otro sistema de control de versiones o algo similar (ver **Chapter 10** sobre tareas de mantenimiento). Habitualmente, usaremos el camino (path) normal por ser casi siempre más rápido.

Para añadir un repositorio local a un proyecto Git existente, puedes usar algo como:

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

Con lo que podrás enviar (push) y recibir (pull) desde dicho remoto exactamente de la misma forma a como lo harías a través de una red.

VENTAJAS

Las ventajas de los repositorios basados en carpetas y archivos, son su simplicidad y el aprovechamiento de los permisos preexistentes de acceso. Si tienes un sistema de archivo compartido que todo el equipo pueda usar, preparar un repositorio es muy sencillo. Simplemente pones el repositorio básico en algún lugar donde todos tengan acceso a él y ajustas los permisos de lectura/escritura según proceda, tal y como lo harías para preparar cualquier otra carpeta compartida. En la próxima sección, “**Configurando Git en un servidor**”, veremos cómo exportar un repositorio básico para conseguir esto.

Este camino es también útil para recuperar rápidamente el contenido del repositorio de trabajo de alguna otra persona. Si tu y otra persona estáis trabajando en el mismo proyecto y ésta quiere mostrarte algo, el usar un comando tal como `git pull /home/john/project` suele ser más sencillo que el que esa persona te lo envie (push) a un servidor remoto y luego tú lo recojas (pull) desde allí.

DESVENTAJAS

La principal desventaja de los repositorios basados en carpetas y archivos es su dificultad de acceso desde distintas ubicaciones. Por ejemplo, si quieres enviar (push) desde tu portátil cuando estás en casa, primero tienes que montar el disco remoto; lo cual puede ser difícil y lento, en comparación con un acceso basado en red.

Cabe destacar también que una carpeta compartida no es precisamente la opción más rápida. Un repositorio local es rápido solamente en aquellas ocasiones en que tienes un acceso rápido a él. Normalmente un repositorio sobre NFS es más lento que un repositorio SSH en el mismo servidor, asumiendo que las pruebas se hacen con Git sobre discos locales en ambos casos.

Protocolos HTTP

Git puede utilizar el protocolo HTTP de dos maneras. Antes de la versión 1.6.6 de Git, solo había una forma de utilizar el protocolo HTTP y normalmente en sólo lectura. Con la llegada de la versión 1.6.6 se introdujo un nuevo protocolo más inteligente que involucra a Git para negociar la transferencia de datos de

una manera similar a como se hace con SSH. En los últimos años, este nuevo protocolo basado en HTTP se ha vuelto muy popular puesto que es más sencillo para el usuario y también más inteligente. Nos referiremos a la nueva versión como el HTTP “Inteligente” y llamaremos a la versión anterior el HTTP “tonto”. Comenzaremos primero con el protocolo HTTP “Inteligente”.

HTTP INTELIGENTE

El protocolo HTTP “Inteligente” funciona de forma muy similar a los protocolos SSH y Git, pero se ejecutan sobre puertos estándar HTTP/S y pueden utilizar los diferentes mecanismos de autenticación HTTP. Esto significa que puede resultar más fácil para los usuarios, puesto que se pueden identificar mediante usuario y contraseña (usando la autenticación básica de HTTP) en lugar de usar claves SSH.

Es, probablemente, la forma más popular de usar Git ahora, puesto que puede configurarse para servir tanto acceso anónimo (como con el protocolo Git) y acceso autenticado para realizar envíos (push), con cifrado similar a como se hace con SSH. En lugar de tener diferentes URL para cada cosa, se puede tener una única URL para todo. Si intentamos subir cambios (push) al repositorio nos pedirá usuario y contraseña, y para accesos de lectura se puede permitir el acceso anónimo o requerir también usuario.

De hecho, para servicios como GitHub, la URL que usamos para ver el repositorio en la web (por ejemplo, “<https://github.com/schacon/simplegit>”) es la misma que usaríamos para clonar y, si tenemos permisos, para enviar cambios.

HTTP TONTO

Si el servidor no dispone del protocolo HTTP “Inteligente”, el cliente de Git intentará con el protocolo clásico HTTP que podemos llamar HTTP “Tonto”. Este protocolo espera obtener el repositorio Git a través de un servidor web como si accediera a ficheros normales. Lo bonito de este protocolo es la simplicidad para configurarlo. Básicamente, todo lo que tenemos que hacer es poner el repositorio Git bajo el directorio raíz de documentos HTTP y especificar un punto de enganche (hook) de post-update (véase “**Puntos de enganche en Git**”). Desde este momento, cualquiera con acceso al servidor web donde se publique el repositorio podrá también clonarlo. Para permitir acceso lectura con HTTP, debes hacer algo similar a lo siguiente:

```
$ 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
```

Y esto es todo. El punto de enganche `post-update` que trae Git de manera predeterminada ejecuta el comando adecuado (`git update-server-info`) para hacer que las operaciones de clonado o recuperación (`fetch`) funcionen de forma adecuada. Este comando se ejecuta cuando se envían cambios (`push`) al repositorio (mediante SSH, por ejemplo); luego, otras personas pueden clonar mediante algo como

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

En este caso concreto, hemos utilizado la carpeta `/var/www/htdocs`, que es el habitual en configuraciones Apache, pero se puede usar cualquier servidor web estático. Basta con que se ponga el repositorio básico (`bare`) en la carpeta correspondiente. Los datos de Git son servidos como ficheros estáticos simples (véase [Chapter 10](#) para saber exactamente cómo se sirven).

Por lo general tendremos que elegir servirlos en lectura/escritura con el servidor HTTP “Inteligente” o en solo lectura con el servidor “tonto”. Mezclar ambos servicios no es habitual.

VENTAJAS

Nos centraremos en las ventajas de la versión “Inteligente” del protocolo HTTP.

La simplicidad de tener una única URL para todos los tipos de acceso y que el servidor pida autenticación solo cuando se necesite hace las cosas muy fáciles para el usuario final. Permitiendo autenticar mediante usuario y contraseña es también una ventaja sobre SSH, ya que los usuarios no tendrán que generar sus claves SSH y subir la pública al servidor antes de comenzar a usarlo. Esta es la principal ventaja para los usuarios menos especializados, o para los usuarios de sistemas donde el SSH no se suele usar. También es un protocolo muy rápido y eficiente, como sucede con el SSH.

También se pueden servir los repositorios en sólo lectura con HTTPS, lo que significa que se puede cifrar la transferencia de datos; incluso se puede identificar a los clientes haciéndoles usar certificados convenientemente firmados.

Otra cosa interesante es que los protocolos HTTP/S son los más ampliamente utilizados, de forma que los cortafuegos corporativos suelen permitir el tráfico a través de esos puertos.

INCONVENIENTES

Git sobre HTTP/S puede ser un poco más complejo de configurar comparado con el SSH en algunos sitios. En otros casos, se adivina poca ventaja sobre el uso de otros protocolos.

Sí utilizamos HTTP para envíos autenticados, proporcionar nuestras credenciales cada vez que se hace puede resultar más complicado que usar claves SSH. Hay, sin embargo, diversas utilidades de cacheo de credenciales, como Keychain en OSX o Credential Manager en Windows; haciendo esto menos incómodo. Lee “**Credential Storage**” para ver cómo configurar el cacheo seguro de contraseñas HTTP en tu sistema.

El Procotolo SSH

SSH es un protocolo muy habitual para alojar repositorios Git en hostings privados. Esto es así porque el acceso SSH viene habilitado de forma predeterminada en la mayoría de los servidores, y si no es así, es fácil de habilitarlo. Además, SSH es un protocolo de red autenticado, y es sencillo de utilizar.

Para clonar un repositorio a través de SSH, puedes indicar una URL ssh:// tal como:

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

También puedes usar la sintaxis estilo scp del protocolo SSH:

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

Pudiendo asimismo prescindir del usuario; en cuyo caso Git asume el usuario con el que estés conectado en ese momento.

VENTAJAS

El uso de SSH tiene múltiples ventajas. En primer lugar, SSH es relativamente fácil de configurar: los demonios (daemons) SSH son de uso común, muchos administradores de red tienen experiencia con ellos y muchas distribuciones del SO los traen predefinidos o tienen herramientas para gestionarlos. Además, el acceso a través de SSH es seguro, estando todas las transferencias encriptadas y autenticadas. Y, por último, al igual que los protocolos HTTP/S, Git y Local, SSH es eficiente, comprimiendo los datos lo más posible antes de transferirlos.

DESVENTAJAS

El aspecto negativo de SSH es su imposibilidad para dar acceso anónimo al repositorio. Todos han de tener configurado un acceso SSH al servidor, incluso aunque sea con permisos de solo lectura; lo que no lo hace recomendable para soportar proyectos abiertos. Si lo usas únicamente dentro de tu red corporativa, posiblemente sea SSH el único protocolo que tengas que emplear. Pero si quieras también habilitar accesos anónimos de solo lectura, tendrás que reservar SSH para tus envíos (push) y habilitar algún otro protocolo para las recuperaciones (pull) de los demás.

El protocolo Git

El protocolo Git es un demonio (daemon) especial, que viene incorporado con Git. Escucha por un puerto dedicado (9418), y nos da un servicio similar al del protocolo SSH; pero sin ningún tipo de autenticación. Para que un repositorio pueda exponerse a través del protocolo Git, tienes que crear en él un archivo *git-daemon-export-ok*; sin este archivo, el demonio no hará disponible el repositorio. Pero, aparte de esto, no hay ninguna otra medida de seguridad. O el repositorio está disponible para que cualquiera lo pueda clonar, o no lo está. Lo cual significa que, normalmente, no se podrá enviar (push) a través de este protocolo. Aunque realmente si que puedes habilitar el envío, si lo haces, dada la total falta de ningún mecanismo de autenticación, cualquiera que encuentre la URL a tu proyecto en Internet, podrá enviar (push) contenidos a él. Ni que decir tiene que esto solo lo necesitarás en contadas ocasiones.

VENTAJAS

El protocolo Git es el más rápido de todos los disponibles. Si has de servir mucho tráfico de un proyecto público o servir un proyecto muy grande, que no requiera autenticación para leer de él, un demonio Git es la respuesta. Utiliza los mismos mecanismos de transmisión de datos que el protocolo SSH, pero sin la sobrecarga de la encriptación ni de la autenticación.

DESVENTAJAS

La pega del protocolo Git, es su falta de autenticación. No es recomendable tenerlo como único protocolo de acceso a tus proyectos. Habitualmente, lo combinarás con un acceso SSH o HTTPS para los pocos desarrolladores con acceso de escritura que envíen (push) material, dejando el protocolo *git://* para los accesos solo-lectura del resto de personas.

Por otro lado, necesita activar su propio demonio, y necesita configurar *xinetd* o similar, lo cual no suele estar siempre disponible en el sistema donde estés trabajando. Requiere además abrir expresamente acceso al puerto 9418 en el cortafuegos, ya que estará cerrado en la mayoría de los cortafuegos corporativos.

Configurando Git en un servidor

Ahora vamos a cubrir la creación de un servicio de Git ejecutando estos protocolos en su propio servidor.

EXAMPLE 4-1.

Aquí demostraremos los comandos y pasos necesarios para hacer las instalaciones básicas simplificadas en un servidor basado en Linux, aunque también es posible ejecutar estos servicios en los servidores Mac o Windows. Configurar un servidor de producción dentro de tu infraestructura sin duda supondrá diferencias en las medidas de seguridad o de las herramientas del sistema operativo, pero se espera que esto le dé la idea general de lo que el proceso involucra.

Para configurar por primera vez un servidor de Git, hay que exportar un repositorio existente en un nuevo repositorio vacío - un repositorio que no contiene un directorio de trabajo. Esto es generalmente fácil de hacer. Para clonar el repositorio con el fin de crear un nuevo repositorio vacío, se ejecuta el comando `clone` con la opción `--bare`. Por convención, los directorios del repositorio vacío terminan en `.git`, así:

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

Deberías tener ahora una copia de los datos del directorio Git en tu directorio `my_project.git`. Esto es más o menos equivalente a algo así:

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

Hay un par de pequeñas diferencias en el archivo de configuración; pero para tu propósito, esto es casi la misma cosa. Toma el repositorio Git en sí mismo, sin un directorio de trabajo, y crea un directorio específicamente para él solo.

Colocando un Repositorio Vacío en un Servidor

Ahora que tienes una copia vacía de tu repositorio, todo lo que necesitas hacer es ponerlo en un servidor y establecer sus protocolos. Digamos que has configurado un servidor llamado `git.example.com` que tiene acceso a SSH, y quieres almacenar todos tus repositorios Git bajo el directorio `/opt/git`. Suponiendo que existe `/opt/git` en ese servidor, puedes configurar tu nuevo repositorio copiando tu repositorio vacío a:

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

En este punto, otros usuarios que con acceso SSH al mismo servidor que tiene permisos de lectura-acceso al directorio `/opt/git` pueden clonar tu repositorio mediante el comando

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

Si un usuario accede por medio de SSH a un servidor y tiene permisos de escritura en el directorio `git my_project.git / opt /`, automáticamente también tendrán acceso push.

Git automáticamente agrega permisos de grupo para la escritura en un repositorio apropiadamente si se ejecuta el comando `git init` con la opción `--shared`.

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

Puedes ver lo fácil que es tomar un repositorio Git, crear una versión vacía, y colocarlo en un servidor al que tú y tus colaboradores tienen acceso SSH. Ahora estan listos para colaborar en el mismo proyecto.

Es importante tener en cuenta que esto es literalmente todo lo que necesitas hacer para ejecutar un útil servidor Git al cual varias personas tendrán acceso - sólo tiene que añadir cuentas con acceso SSH a un servidor, y subir un repositorio vacío en alguna parte a la que todos los usuarios puedan leer y escribir. Estás listo para trabajar. Nada más es necesario.

En las próximas secciones, verás cómo ampliar a configuraciones más sofisticadas. Esta sección incluirá no tener que crear cuentas para cada usuario, añadiendo permisos de lectura pública a los repositorios, la creación de interfaces de usuario web y más. Sin embargo, ten en cuenta que para colaborar con

un par de personas en un proyecto privado, todo_lo_que_necesitas_es un servidor SSH y un repositorio vacío.

Pequeñas configuraciones

Si tienes un pequeño equipo o acabas de probar Git en tu organización y tienes sólo unos pocos desarrolladores, las cosas pueden ser simples para tí. Uno de los aspectos más complicados de configurar un servidor Git es la gestión de usuarios. Si quieres que algunos repositorios sean de sólo lectura para ciertos usuarios y lectura / escritura para los demás, el acceso y los permisos pueden ser un poco más difíciles de organizar.

ACCESO SSH

Si tienes un servidor al que todos los desarrolladores ya tienen acceso SSH, es generalmente más fácil de configurar el primer repositorio allí, porque no hay que hacer casi ningún trabajo (como ya vimos en la sección anterior). Si quieres permisos de acceso más complejas en tus repositorios, puedes manejarlos con los permisos del sistema de archivos normales del sistema operativo donde tu servidor se ejecuta.

Si deseas colocar los repositorios en un servidor que no tiene cuentas para todo el mundo en su equipo para el que deseas tener acceso de escritura, debes configurar el acceso SSH para ellos. Suponiendo que tienes un servidor con el que hacer esto, ya tiene un servidor SSH instalado, y así es como estás accediendo al servidor.

Hay algunas maneras con las cuales puedes dar acceso a todo tu equipo. La primera es la creación de cuentas para todo el mundo, que es sencillo, pero puede ser engorroso. Podrías no desear ejecutar `adduser` y establecer contraseñas temporales para cada usuario.

Un segundo método consiste en crear un solo usuario *git* en la máquina, preguntar a cada usuario de quién se trata para otorgarle permisos de escritura para que te envíe una llave SSH pública, y agregar esa llave al archivo `~ / .ssh / authorized_keys` de tu nuevo usuario *git*. En ese momento, todo el mundo podrá acceder a esa máquina mediante el usuario *git*. Esto no afecta a los datos commit de ninguna manera - el usuario SSH con el que te conectas no puede modificar los commits que has registrado.

Otra manera de hacer que tu servidor SSH autentifique desde un servidor LDAP o desde alguna otra fuente de autenticación centralizada que pudieras tener ya configurada. Mientras que cada usuario sea capaz de tener acceso shell a la máquina, cualquier mecanismo de autenticación SSH que se te ocurra debería de funcionar.

Generando tu clave pública SSH

Tal y como se ha comentado, muchos servidores Git utilizan la autentificación a través de claves públicas SSH. Y, para ello, cada usuario del sistema ha de generarse una, si es que no la tiene ya. El proceso para hacerlo es similar en casi cualquier sistema operativo. Ante todo, asegurarte que no tengas ya una clave. Por defecto, las claves de cualquier usuario SSH se guardan en la carpeta `~/.ssh` de dicho usuario. Puedes verificar si tienes ya unas claves, simplemente situandote sobre dicha carpeta y viendo su contenido:

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

Has de buscar un par de archivos con nombres tales como *algo* y *algo.pub*; siendo ese “algo” normalmente *id_dsa* o *id_rsa*. El archivo terminado en *.pub* es tu clave pública, y el otro archivo es tu clave privada. Si no tienes esos archivos (o no tienes ni siquiera la carpeta `.ssh`), has de crearlos; utilizando un programa llamado *ssh-keygen*, que viene incluido en el paquete SSH de los sistemas Linux/Mac o en el paquete MSysGit en los sistemas Windows:

```
$ ssh-keygen
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
```

Como se ve, este comando primero solicita confirmación de dónde van a a guardarse las claves (`.ssh/id_rsa`), y luego solicita, dos veces, una contraseña (passphrase), contraseña que puedes dejar en blanco si no deseas tener que teclearla cada vez que uses la clave.

Tras generarla, cada usuario ha de encargarse de enviar su clave pública a quienquiera que administre el servidor Git (en el caso de que éste esté configurado con SSH y así lo requiera). Esto se puede realizar simplemente copiando los contenidos del archivo terminado en `.pub` y enviandoselos por correo elec-

trónico. La clave pública será una serie de números, letras y signos, algo así como esto:

```
$ cat ~/.ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAQABAAQCB007n/ww+ouN4gSLKssMxXnB0vf9LGt4L
ojG6rs6hPB09j9R/T17/x4lhJA0F3FR1rP6kYBRsWj2aThGw6HXLm9/5zytK6Ztg3RPKK+4k
Yjh6541NYsnEAZuXz0jTTyAUfrtU3Z5E003C4ox0j6H0rfIF1kKI9MAQLMdGW1GYEIgS9Ez
Sdfd8AccIicTDWbqlAcU4UpkaX8KyGlLwsNuuGztobF8m72ALC/nLF6JLtpOfwFBlgc+myiv
07TCUSbdLQlgMVOFq1I2uPWQ0kOWQAHuke0mfjy2jctxSDBQ220ymjaNsHT4kgtZg2AYYgPq
dAv8JggJICUvax2T9va5 gsg-keypair
```

Para más detalles sobre cómo crear unas claves SSH en variados sistemas operativos, consultar la correspondiente guía en GitHub: <https://help.github.com/articles/generating-ssh-keys>.

Configurando el servidor

Vamos a avanzar en los ajustes de los accesos SSH en el lado del servidor. En este ejemplo, usarás el método de las `authorized_keys` (claves autorizadas) para autenticar a tus usuarios. Se asume que tienes un servidor en marcha, con una distribución estandar de Linux, tal como Ubuntu. Comienzas creando un usuario `git` y una carpeta `.ssh` para él.

```
$ sudo adduser git
$ su git
$ cd
$ mkdir .ssh && chmod 700 .ssh
$ touch .ssh/authorized_keys && chmod 600 .ssh/authorized_keys
```

Y a continuación añades las claves públicas de los desarrolladores al archivo `authorized_keys` del usuario `git` que has creado. Suponiendo que hayas recibido las claves por correo electrónico y que las has guardado en archivos temporales. Y recordando que las claves públicas son algo así como:

```
$ cat /tmp/id_rsa.john.pub
ssh-rsa AAAAB3NzaC1yc2EAAAQABAAQCB007n/ww+ouN4gSLKssMxXnB0vf9LGt4L
ojG6rs6hPB09j9R/T17/x4lhJA0F3FR1rP6kYBRsWj2aThGw6HXLm9/5zytK6Ztg3RPKK+4k
Yjh6541NYsnEAZuXz0jTTyAUfrtU3Z5E003C4ox0j6H0rfIF1kKI9MAQLMdGW1GYEIgS9Ez
Sdfd8AccIicTDWbqlAcU4UpkaX8KyGlLwsNuuGztobF8m72ALC/nLF6JLtpOfwFBlgc+myiv
07TCUSbdLQlgMVOFq1I2uPWQ0kOWQAHuke0mfjy2jctxSDBQ220ymjaNsHT4kgtZg2AYYgPq
dAv8JggJICUvax2T9va5 gsg-keypair
```

No tienes más que añadirlas al archivo `authorized_keys` dentro del directorio `.ssh`:

```
$ 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
```

Tras esto, puedes preparar un repositorio básico vacío para ellos, usando el comando `git init` con la opción `--bare` para inicializar el repositorio sin carpeta de trabajo:

```
$ cd /opt/git
$ mkdir project.git
$ cd project.git
$ git init --bare
Initialized empty Git repository in /opt/git/project.git/
```

Y John, Josie o Jessica podrán enviar (push) la primera versión de su proyecto a dicho repositorio, añadiéndolo como remoto y enviando (push) una rama (branch). Cabe indicar que alguien tendrá que iniciar sesión en la máquina y crear un repositorio básico, cada vez que se desee añadir un nuevo proyecto. Suponiendo, por ejemplo, que se llame `gitserver` el servidor donde has puesto el usuario `git` y los repositorios; que dicho servidor es interno a vuestra red y que está asignado el nombre `gitserver` en vuestro DNS. Podrás utilizar comandos tales como (suponiendo que `myproject` es un proyecto ya creado con algunos ficheros):

```
# on Johns computer
$ cd myproject
$ git init
$ git add .
$ git commit -m 'initial commit'
$ git remote add origin git@gitserver:/opt/git/project.git
$ git push origin master
```

Tras lo cual, otros podrán clonarlo y enviar cambios de vuelta:

```
$ git clone git@gitserver:/opt/git/project.git
$ cd project
$ vim README
```

```
$ git commit -am 'fix for the README file'
$ git push origin master
```

Con este método, puedes preparar rápidamente un servidor Git con acceso de lectura/escritura para un grupo de desarrolladores.

Observa que todos esos usuarios pueden también entrar en el servidor obteniendo un intérprete de comandos con el usuario `git`. Si quieres restringirlo, tendrás que cambiar el intérprete (shell) en el fichero `passwd`.

Para una mayor protección, puedes restringir fácilmente el usuario `git` a realizar solamente actividades relacionadas con Git, utilizando un shell limitado llamado `git-shell`, que viene incluido en Git. Si lo configuras como el shell de inicio de sesión de tu usuario `git`, dicho usuario no tendrá acceso al shell normal del servidor. Para especificar el `git-shell` en lugar de `bash` o de `csh` como el shell de inicio de sesión de un usuario, has de editar el archivo `/etc/passwd`:

```
$ cat /etc/shells  # mirar si `git-shell` ya está aquí. Si no...
$ which git-shell  # buscar `git-shell` en nuestro sistema
$ sudo vim /etc/shells # y añadirlo al final de este fichero con el camino (path)
```

Ahora ya puedes cambiar la shell del usuario utilizando `chsh <username>`:

```
$ sudo chsh git # poner aquí la nueva shell, normalmente será: /usr/bin/git-shell
```

De esta forma dejamos al usuario `git` limitado a utilizar la conexión SSH solamente para enviar (push) y recibir (pull) repositorios, sin posibilidad de iniciar una sesión normal en el servidor. Si pruebas a hacerlo, recibirás un rechazo de inicio de sesión:

```
$ 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.
```

Los comandos remotos de Git funcionarán con normalidad, pero los usuarios no podrán obtener un intérprete de comandos del sistema. Tal como nos avisa, también se puede establecer un directorio llamado `git-shell-commands` en la cuenta del usuario `git` para personalizar un poco el `git-shell`. Por ejemplo, se puede restringir qué comandos de Git se aceptarán o se puede

personalizar el mensaje que los usuarios verán si intentan abrir un intérprete de comandos con SSH.

Ejecutando `git help shell` veremos más información sobre cómo personalizar el shell.

El demonio Git

Ahora vamos a configurar un demonio sirviendo repositorios mediante el protocolo “Git”. Es la forma más común para dar acceso anónimo pero rápido a los repositorios. Recuerda que puesto que es un acceso no autenticado, todo lo que sirvas mediante este protocolo será público en la red.

Si activas el protocolo en un servidor más allá del cortafuegos, lo debes usar únicamente en proyectos que deban ser visibles a todo el mundo. Si el servidor está detrás de un cortafuegos, puedes usarlo en proyectos a los que un gran número de personas o de ordenadores (por ejemplo, servidores de integración continua o de compilación) tengan acceso de sólo lectura y no necesiten establecer una clave SSH para cada uno de ellos.

El protocolo Git es relativamente fácil de configurar. Básicamente, necesitas ejecutar el comando con la variante demonio (daemon):

```
git daemon --reuseaddr --base-path=/opt/git/ /opt/git/
```

El parámetro `--reuseaddr` permite al servidor reiniciarse sin esperar a que se liberen viejas conexiones; el parámetro `--base-path` permite a los usuarios clonar proyectos sin necesidad de indicar su camino completo; y el camino indicado al final del comando mostrará al demonio Git dónde buscar los repositorios a exportar. Si tienes un cortafuegos activo, necesitarás abrir el puerto 9418 para la máquina donde estás configurando el demónio Git.

Este proceso se puede demonizar de diferentes maneras, dependiendo del sistema operativo con el que trabajas. En una máquina Ubuntu, puedes usar un script de arranque. Poniendo en el siguiente archivo:

```
/etc/event.d/local-git-daemon
```

un script tal como:

```
start on startup
stop on shutdown
exec /usr/bin/git daemon \
--user=git --group=git \
```

```
--reuseaddr \
--base-path=/opt/git/ \
/opt/git/
respawn
```

Por razones de seguridad, es recomendable lanzar este demonio con un usuario que tenga únicamente permisos de lectura en los repositorios (Lo puedes hacer creando un nuevo usuario *git-ro* y lanzando el demonio con él). Para simplificar, en estos ejemplos vamos a lanzar el demonio Git bajo el mismo usuario *git* que se usa con *git-shell*.

Tras reiniciar tu máquina, el demonio Git arrancará automáticamente y se reiniciará cuando se caiga. Para arrancarlo sin necesidad de reiniciar la máquina, puedes utilizar el comando:

```
initctl start local-git-daemon
```

En otros sistemas operativos, puedes utilizar *xinetd*, un script en el sistema *sysvinit*, o alguna otra manera (siempre y cuando demonices el comando y puedas monitorizarlo).

A continuación, has de indicar a Git a cuáles de tus repositorios ha de permitir acceso sin autenticar. Lo puedes hacer creando en cada repositorio un fichero llamado *git-daemon-export-ok*.

```
$ cd /path/to/project.git
$ touch git-daemon-export-ok
```

La presencia de este fichero dice a Git que este proyecto se puede servir sin problema sin necesidad de autenticación de usuarios.

HTTP Inteligente

Ahora ya tenemos acceso autenticado mediante SSH y anónimo mediante *git://*, pero hay también otro protocolo que permite tener ambos accesos a la vez. Configurar HTTP inteligente consiste, básicamente, en activar en el servidor web un script CGI que viene con Git, llamado *git-http-backend*. Este CGI leerá la ruta y las cabeceras enviadas por los comandos *git fetch* o *git push* a una URL de HTTP y determinará si el cliente puede comunicar con HTTP (lo que será cierto para cualquier cliente a partir de la versión 1.6.6). Si el CGI comprueba que el cliente es inteligente, comunicará inteligentemente con él;

en otro caso pasará a usar el comportamiento tonto (es decir, es compatible con versiones más antiguas del cliente).

Revisemos una configuración básica. Pondremos Apache como servidor de CGI. Si no tienes Apache configurado, lo puedes instalar en un Linux con un comando similar a este:

```
$ sudo apt-get install apache2 apache2-utils
$ a2enmod cgi alias env
```

Esto además activa los módulos `mod_cgi`, `mod_alias`, y `mod_env`, que van a hacer falta para que todo esto funcione.

A continuación tenemos que añadir algunas cosas a la configuración de Apache para que se utilice `git-http-backend` para cualquier cosa que haya bajo la carpeta virtual `/git`.

```
SetEnv GIT_PROJECT_ROOT /opt/git
SetEnv GIT_HTTP_EXPORT_ALL
ScriptAlias /git/ /usr/libexec/git-core/git-http-backend/
```

Si dejas sin definir la variable de entorno `GIT_HTTP_EXPORT_ALL`, Git solo servirá a los clientes anónimos aquellos repositorios que contengan el fichero `daemon-export-ok`, igual que hace el demonio Git.

Ahora tienes que decirle a Apache que acepte peticiones en esta ruta con algo similar a esto:

```
<Directory "/usr/lib/git-core*">
    Options ExecCGI Indexes
    Order allow,deny
    Allow from all
    Require all granted
</Directory>
```

Finalmente, si quieres que los clientes autenticados tengan acceso de escritura, tendrás que crear un bloque Auth similar a este:

```
<LocationMatch "^/git/.*/git-receive-pack$">
    AuthType Basic
    AuthName "Git Access"
    AuthUserFile /opt/git/.htpasswd
```

```
Require valid-user  
</LocationMatch>
```

Esto requiere que hagas un fichero `.htaccess` que contenga las contraseñas cifradas de todos los usuarios válidos. Por ejemplo, para añadir el usuario “schacon” a este fichero:

```
$ htdigest -c /opt/git/.htpasswd "Git Access" schacon
```

Hay un montón de maneras de dar acceso autenticado a los usuarios con Apache, y tienes que elegir una. Esta es la forma más simple de hacerlo. Probablemente también te interese hacerlo todo con SSL para que todos los datos vayan cifrados.

No queremos profundizar en los detalles de la configuración de apache, ya que puedes tener diferentes necesidades de autentificación o querer utilizar un servidor diferente. La idea es que Git trae un CGI llamado `git-http-backend` que cuando es llamado, hace toda la negociación y envío o recepción de datos a través de HTTP. Por sí mismo no implementa autenticación de ningún tipo, pero puede controlarse desde el servidor web que lo utiliza. Puedes configurar esto en casi cualquier servidor web que pueda trabajar con CGI, el que más te guste.

Para más información sobre cómo configurar Apache, mira la documentación: <http://httpd.apache.org/docs/current/howto/auth.html>

GitWeb

Ahora que ya tienes acceso básico de lectura/escritura y de solo-lectura a tu proyecto, puedes querer instalar un visualizador web. Git trae un script CGI, denominado `GitWeb`, que es el que usaremos para este propósito.

The screenshot shows the GitWeb interface for a repository named 'summary'. At the top, there's a navigation bar with links for 'projects / .git / summary', 'summary', 'shortlog', 'log', 'commit', 'commitsdiff', and 'tree'. Below the navigation is a search bar with fields for 'commit' and 'search:' and a 're' button. The main content area is divided into sections: 'shortlog' (listing commits from June 2014), 'tags' (listing tags from 3 weeks ago to 3 years ago), and other sections like 'description', 'owner', and 'last change' which are currently empty.

FIGURE 4-1

The GitWeb web-based user interface.

Si quieres comprobar cómo podría quedar GitWeb con tu proyecto, Git dispone de un comando para activar una instancia temporal, si en tu sistema tienes un servidor web ligero, como por ejemplo `lighttpd` o `webrick`. En las máquinas Linux, `lighttpd` suele estar habitualmente instalado, por lo que tan solo has de activarlo lanzando el comando `git instaweb`, estando en la carpeta de tu proyecto. Si tienes una máquina Mac, Leopard trae preinstalado Ruby, por lo que `webrick` puede ser tu mejor apuesta. Para instalar `instaweb` disponiendo de un controlador no-`lighttpd`, puedes lanzarlo con la opción `--httpd`.

```
$ git instaweb --httpd=webrick
[2009-02-21 10:02:21] INFO  WEBrick 1.3.1
[2009-02-21 10:02:21] INFO  ruby 1.8.6 (2008-03-03) [universal-darwin9.0]
```

Esto arranca un servidor HTTPD en el puerto 1234, y luego arranca un navegador que abre esa página. Es realmente sencillo. Cuando ya hayas terminado y quieras apagar el servidor, puedes lanzar el mismo comando con la opción `--stop`:

```
$ git instaweb --httpd=webrick --stop
```

Siquieres disponer permanentemente de un interface web para tu equipo o para un proyecto de código abierto que albergues, necesitarás ajustar el script CGI para ser servido por tu servidor web habitual. Algunas distribuciones Linux suelen incluir el paquete `gitweb`, y podrás instalarlo a través de las utilidades `apt` o `yum`; merece la pena probarlo en primer lugar. Enseguida vamos a revisar el proceso de instalar GitWeb manualmente. Primero, necesitas el código fuente de Git, que viene con GitWeb, para generar un script CGI personalizado:

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
$ cd git/
$ make GITWEB_PROJECTROOT="/opt/git" prefix=/usr gitweb
      SUBDIR gitweb
      SUBDIR ../
make[2]: `GIT-VERSION-FILE' is up to date.
      GEN gitweb.cgi
      GEN static/gitweb.js
$ sudo cp -Rf gitweb /var/www/
```

Fíjate que es necesario indicar la ubicación donde se encuentran los repositorios Git, utilizando la variable `GITWEB_PROJECTROOT`. A continuación, tienes que preparar Apache para que utilice dicho script. Para ello, puedes añadir un `VirtualHost`:

```
<VirtualHost *:80>
    ServerName gitserver
    DocumentRoot /var/www/gitweb
    <Directory /var/www/gitweb>
        Options ExecCGI +FollowSymLinks +SymLinksIfOwnerMatch
        AllowOverride All
        order allow,deny
        Allow from all
        AddHandler cgi-script cgi
        DirectoryIndex gitweb.cgi
    </Directory>
</VirtualHost>
```

Recordar una vez más que GitWeb puede servirse desde cualquier servidor web con capacidades CGI o Perl. Por lo que si prefieres utilizar algún otro, no debería ser difícil configurarlo. En este momento, deberías poder visitar <http://gitserver/> para ver tus repositorios online.

GitLab

GitWeb es muy simple. Si buscas un servidor Git más moderno, con todas las funciones, tienes algunas soluciones de código abierto que puedes utilizar en su lugar. Puesto que GitLab es una de las más populares, vamos a ver aquí cómo se instala y se usa, a modo de ejemplo. Es algo más complejo que GitWeb y requiere algo más de mantenimiento, pero es una opción con muchas más funciones.

Instalación

GitLab es una aplicación web con base de datos, por lo que su instalación es algo más complicada. Por suerte, es un proceso muy bien documentado y soportado.

Hay algunos métodos que puedes seguir para instalar GitLab. Para tener algo rápidamente, puedes descargar una máquina virtual o un instalador one-click desde <https://bitnami.com/stack/gitlab>, y modificar la configuración para tu caso particular. La pantalla de inicio de Bitnami (a la que se accede con alt-→); te dirá la dirección IP y el usuario y contraseña utilizados para instalar GitLab.

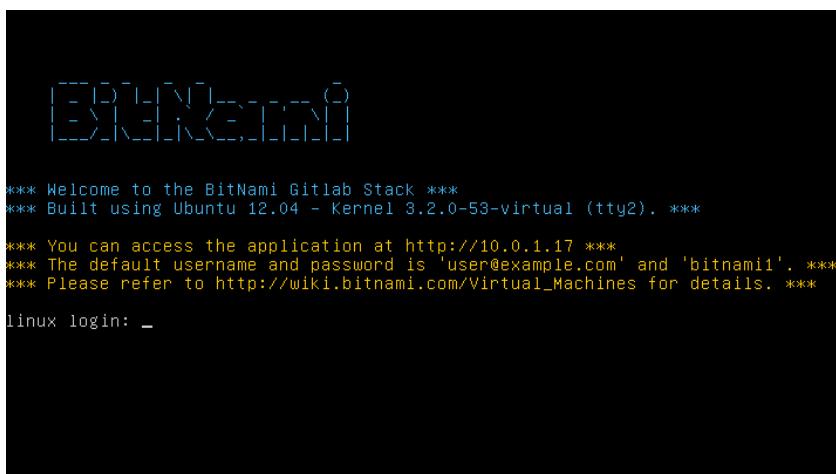


FIGURE 4-2

Página de login de la máquina virtual Bitnami.

Para las demás cosas, utiliza como guía los ficheros `readme` de la edición Community de GitLab, que se pueden encontrar en <https://gitlab.com/gitlab-org/gitlab-ce/tree/master>. Aquí encontrarás ayuda para instalar GitLab usando

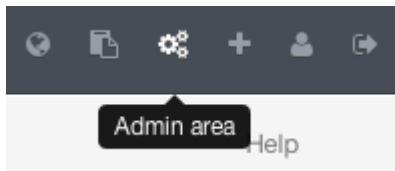
recetas Chef, una máquina virtual para Digital Ocean, y paquetes RPM y DEB (los cuales, en el momento de escribir esto, aun estaban en beta). También hay guías “no oficiales” para configurar GitLab en sistemas operativos o con bases de datos no estándar, un script de instalación completamente manual y otros muchos temas.

Administración

La interfaz de administración de GitLab se accede mediante la web. Simplemente abre en tu navegador la IP o el nombre de máquina donde has instalado Gitlab, y entra con el usuario administrador. El usuario predeterminado es `admin@local.host`, con la contraseña `5iveL!fe` (que te pedirá cambiar cuando entres por primera vez). Una vez dentro, pulsa en el ícono “Admin area” del menú superior derecho.

FIGURE 4-3

El ícono “Admin area” del menú de GitLab.



USUARIOS

Los usuarios en Gitlab son las cuentas que abre la gente. Las cuentas de usuario no tienen ninguna complicación: viene a ser una colección de información personal unida a la información de login. Cada cuenta tiene un **espacio de nombres** (namespace) que es una agrupación lógica de los proyectos que pertenecen al usuario. De este modo, si el usuario `jane` tiene un proyecto llamado `project`, la URL de ese proyecto sería `http://server/jane/project`.

FIGURE 4-4

Pantalla de administración de usuarios en GitLab.

Tenemos dos formas de borrar usuarios. “Bloquear” un usuario evita que el usuario entre en Gitlab, pero los datos de su espacio de nombres se conservan, y los commits realizados por el usuario seguirán a su nombre y relacionados con su perfil.

“Destruir” un usuario, por su parte, borra completamente al usuario de la base de datos y el sistema de ficheros. Todos los proyectos y datos de su espacio de nombres se perderá, así como cualquier grupo que le pertenezca. Esto es, por supuesto, la acción más permanente y destructiva, y casi nunca se usa.

GRUPOS

Un grupo de GitLab es un conjunto de proyectos, junto con los datos acerca de los usuarios que tienen acceso. Cada grupo tiene también un espacio de nombres específico (al igual que los usuarios). Por ejemplo, si el grupo **formacion** tuviese un proyecto **materiales** su URL sería: <http://server/formacion/materiales>.

FIGURE 4-5

Pantalla de administración de grupos en GitLab.

Cada grupo se asocia con un conjunto de usuarios, y cada usuario tiene un nivel de permisos sobre los proyectos así como el propio grupo. Estos permisos van desde el de “Invitado” (que solo permite manejar incidencias y chat) hasta el de “Propietario” (con control absoluto del grupo, sus miembros y sus proyectos). Los tipos de permisos son muy numerosos para detallarlos aquí, pero en la ayuda de la pantalla de administración de GitLab la encontraremos fácilmente.

PROYECTOS

Un proyecto en GitLab corresponde con un repositorio Git. Cada proyecto pertenece a un espacio de nombres, bien sea de usuario o de grupo. Si el proyecto pertenece a un usuario, el propietario del mismo tendrá control directo sobre quién tiene acceso al proyecto; si el proyecto pertenece a un grupo, los permisos de acceso por parte de los usuarios estarán también determinados por los niveles de acceso de los miembros del grupo.

Cada proyecto tiene también un nivel de visibilidad, que controla quién tiene acceso de lectura a las páginas del proyecto y al propio repositorio. Si un proyecto es *Privado*, el propietario debe conceder los accesos para que determinados usuarios tengan permisos. Un proyecto *Interno* es visible a cualquier usuario identificado, y un proyecto *Público* es visible a todos, incluso usuarios identificados y visitantes. Observa que esto controla también el acceso de lectura git (“fetch”) así como el acceso a la página web del proyecto.

ENGANCHES (HOOKS)

GitLab tiene soporte para los enganches (hooks), tanto a nivel de proyecto como del sistema. Para cualquiera de ellos, el servidor GitLab realizará una peti-

ción HTTP POST con determinados datos JSON cuando ocurran determinados eventos. Es una manera interesante de conectar los repositorios y la instancia de GitLab con el resto de los mecanismos automáticos de desarrollo, como servidores de integración continua (CI), salas de charla y otras utilidades de despliegue.

Uso básico

Lo primero que tienes que hacer en GitLab es crear un nuevo proyecto. Esto lo consigues pulsando el icono “+” en la barra superior. Te preguntará por el nombre del proyecto, el espacio de nombres al que pertenece y qué nivel de visibilidad debe tener. Esta información, en su mayoría, no es fija y puedes cambiarla más tarde en la pantalla de ajustes. Pulsa en “Create Project” y habrás terminado.

Una vez que tengas el proyecto, querrás usarlo para un repositorio local de Git. Cada proyecto se puede acceder por HTTPS o SSH, protocolos que podemos configurar en nuestro repositorio como un Git remoto. La URL la encontrarás al principio de la página principal del proyecto. Para un repositorio local existente, puedes crear un remoto llamado `gitlab` del siguiente modo:

```
$ git remote add gitlab https://server/namespace/project.git
```

Si no tienes copia local del repositorio, puedes hacer esto:

```
$ git clone https://server/namespace/project.git
```

La interfaz web te permite acceder a diferentes vistas interesantes del repositorio. Además la página principal del proyecto muestra la actividad reciente, así como enlaces que permiten acceder a los ficheros del proyecto y a los diferentes commits.

Trabajando con GitLab

Para trabajar en un proyecto GitLab lo más simple es tener acceso de escritura (push) sobre el repositorio git. Puedes añadir usuarios al proyecto en la sección “Members” de los ajustes del mismo, y asociar el usuario con un nivel de acceso (los niveles los hemos visto en “**Grupos**”). Cualquier nivel de acceso tipo “Developer” o superior permite al usuario enviar commits y ramas sin ninguna limitación.

Otra forma, más desacoplada, de colaboración, es mediante las peticiones de fusión (merge requests). Esta característica permite a cualquier usuario con acceso de lectura, participar de manera controlada. Los usuarios con acceso directo pueden, simplemente, crear la rama, enviar commits y luego abrir una petición de fusión desde su rama hacia la rama `master` u otra cualquiera. Los usuarios sin permiso de push pueden hacer un “fork” (es decir, su propia copia del repositorio), enviar sus cambios a *esa copia*, y abrir una petición de fusión desde su fork hacia el proyecto del que partió. Este modelo permite al propietario tener un control total de lo que entra en el repositorio, permitiendo sin embargo la cooperación de usuarios en los que no se confía el acceso total.

Las peticiones de fusión y las incidencias (issues) son las principales fuentes de discusión en los proyectos de GitLab. Cada petición de fusión permite una discusión sobre el cambio propuesto (similar a una revisión de código), así como un hilo de discusión general. Ambas pueden asignarse a usuarios, o ser organizadas en hitos (milestones).

Esta sección se ha enfocado principalmente hacia las características de GitLab relacionadas con Git, pero como proyecto ya maduro, tiene muchas otras características para ayudar en la coordinación de grupos de trabajo, como wikis de proyecto y utilidades de mantenimiento. Una ventaja de GitLab es que, una vez que el servidor está configurado y funcionando, rara vez tendrás que tocar un fichero de configuración o acceder al servidor mediante SSH; casi toda la administración y uso se realizará mediante el navegador web.

Git en un alojamiento externo

Si no quieres realizar todo el trabajo que implica poner en marcha tu propio servidor Git, tienes varias opciones para alojar tus proyectos Git en un sitio externo dedicado. Esto tiene varias ventajas: normalmente en los alojamientos externos es fácil configurar y comenzar proyectos y no hay que preocuparse del mantenimiento del servidor o de su monitorización. Aunque pongas en marcha tu propio servidor internamente, probablemente quieras usar un sitio público para tu código abierto. Será más fácil que la comunidad de software libre encuentre tu proyecto y colabore.

Actualmente hay bastantes opciones de alojamiento para elegir, cada una con sus ventajas e inconvenientes. Para ver una lista actualizada, mira la página acerca de alojamiento Git en el wiki principal de Git, en <https://git.wiki.kernel.org/index.php/GitHosting>

Nos ocuparemos en detalle de Github en [Chapter 6](#), al ser el sitio de alojamiento de proyectos más grande, y donde probablemente encuentres otros proyectos en los que quieras participar, pero en cualquier caso hay docenas de sitios para elegir sin necesidad de configurar tu propio servidor Git.

Resumen

Tienes varias opciones para obtener un repositorio Git remoto y ponerlo a funcionar para que puedas colaborar con otras personas o compartir tu trabajo.

Mantener tu propio servidor te da control y te permite correr tu servidor dentro de tu propio cortafuegos, pero tal servidor generalmente requiere una importante cantidad de tu tiempo para configurar y mantener. Si almacenas tus datos en un servidor hospedado, es fácil de configurar y mantener; sin embargo, tienes que ser capaz de mantener tu código en los servidores de alguien más, y algunas organizaciones no te lo permitirán.

Debería ser un proceso claro determinar que solución o combinación de soluciones es apropiada para tí y para tu organización.

5

Git en entornos distribuidos

Ahora que ya tienes un repositorio Git configurado como punto de trabajo para que los desarrolladores compartan su código, y además ya conoces los comandos básicos de Git para usar en local, verás cómo se puede utilizar alguno de los flujos de trabajo distribuido que Git permite.

En este capítulo verás como trabajar con Git en un entorno distribuido como colaborador o como integrador. Es decir, aprenderás como contribuir adecuadamente a un proyecto, de manera fácil tanto para tí como para el responsable del proyecto, y también como mantener adecuadamente un proyecto con múltiples desarrolladores.

Flujos de trabajo distribuidos

A diferencia de Sistemas Centralizados de Control de Versiones (CVCSs, Centralized Version Control Systems), la naturaleza distribuido de Git te permite mucha más flexibilidad en la manera de colaborar en proyectos. En los sistemas centralizados, cada desarrollador es un nodo de trabajo más o menos en igualdad con un repositorio central. En Git, sin embargo, cada desarrollador es potencialmente un nodo o un repositorio - es decir, cada desarrollador puede contribuir a otros repositorios y mantener un repositorio público en el cual otros pueden basar su trabajo y al cual pueden contribuir.

Esto abre un enorme rango de posibles flujos de trabajo para tu proyecto y/o tu equipo, así que revisaremos algunos de los paradigmas que toman ventajas de esta flexibilidad. Repasaremos las fortalezas y posibles debilidades de cada diseño; podrás elegir uno solo o podrás mezclarlos para escoger características concretas de cada uno.

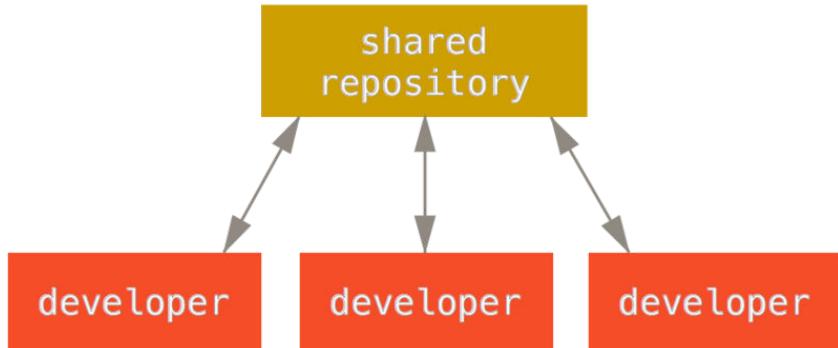
Flujos de trabajo centralizado

En sistemas centralizados, habitualmente solo hay un modelo de colaboración - el flujo de trabajo centralizado. Un repositorio o punto central que acepta có-

digo y todos sincronizan su trabajo con él. Unos cuantos desarrolladores son nodos de trabajo - consumidores de dicho repositorio - y sincronizan con ese punto.

FIGURE 5-1

Centralized workflow.



Esto significa que si dos desarrolladores clonian desde el punto central, y ambos hacen cambios, solo el primer desarrollador en subir sus cambios lo podrá hacer sin problemas. El segundo desarrollador debe fusionar el trabajo del primero antes de subir sus cambios, para no sobreescribir los cambios del primero. Este concepto es válido tanto en Git como en Subversion. This concept is as true in Git as it is in Subversion (or any CVCS), and this model works perfectly well in Git.

If you are already comfortable with a centralized workflow in your company or team, you can easily continue using that workflow with Git. Simply set up a single repository, and give everyone on your team push access; Git won't let users overwrite each other. Say John and Jessica both start working at the same time. John finishes his change and pushes it to the server. Then Jessica tries to push her changes, but the server rejects them. She is told that she's trying to push non-fast-forward changes and that she won't be able to do so until she fetches and merges. This workflow is attractive to a lot of people because it's a paradigm that many are familiar and comfortable with.

This is also not limited to small teams. With Git's branching model, it's possible for hundreds of developers to successfully work on a single project through dozens of branches simultaneously.

Integration-Manager Workflow

Because Git allows you to have multiple remote repositories, it's possible to have a workflow where each developer has write access to their own public

repository and read access to everyone else's. This scenario often includes a canonical repository that represents the “official” project. To contribute to that project, you create your own public clone of the project and push your changes to it. Then, you can send a request to the maintainer of the main project to pull in your changes. The maintainer can then add your repository as a remote, test your changes locally, merge them into their branch, and push back to their repository. The process works as follows (see **Figure 5-2**):

1. The project maintainer pushes to their public repository.
2. A contributor clones that repository and makes changes.
3. The contributor pushes to their own public copy.
4. The contributor sends the maintainer an e-mail asking them to pull changes.
5. The maintainer adds the contributor's repo as a remote and merges locally.
6. The maintainer pushes merged changes to the main repository.

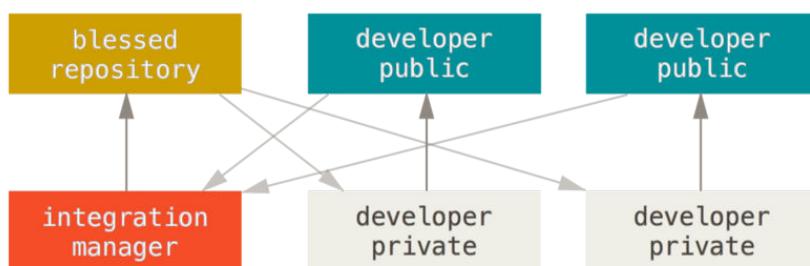


FIGURE 5-2
Integration-manager workflow.

This is a very common workflow with hub-based tools like GitHub or GitLab, where it's easy to fork a project and push your changes into your fork for everyone to see. One of the main advantages of this approach is that you can continue to work, and the maintainer of the main repository can pull in your changes at any time. Contributors don't have to wait for the project to incorporate their changes – each party can work at their own pace.

Dictator and Lieutenants Workflow

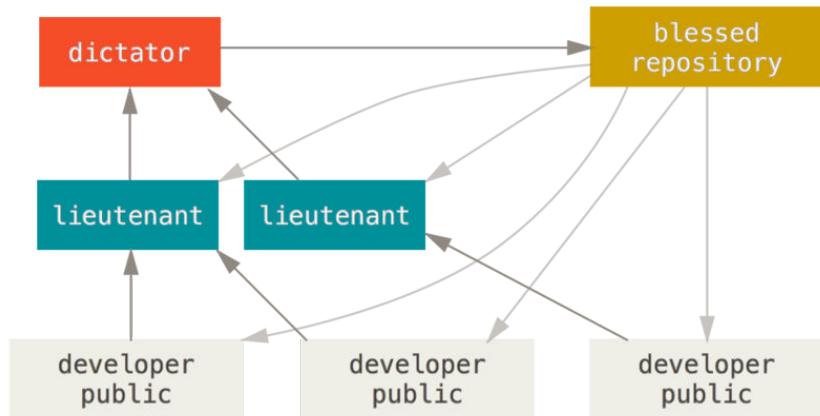
This is a variant of a multiple-repository workflow. It's generally used by huge projects with hundreds of collaborators; one famous example is the Linux kernel. Various integration managers are in charge of certain parts of the reposi-

ry; they're called lieutenants. All the lieutenants have one integration manager known as the benevolent dictator. The benevolent dictator's repository serves as the reference repository from which all the collaborators need to pull. The process works like this (see **Figure 5-3**):

1. Regular developers work on their topic branch and rebase their work on top of `master`. The `master` branch is that of the dictator.
2. Lieutenants merge the developers' topic branches into their `master` branch.
3. The dictator merges the lieutenants' `master` branches into the dictator's `master` branch.
4. The dictator pushes their `master` to the reference repository so the other developers can rebase on it.

FIGURE 5-3

Benevolent dictator workflow.



This kind of workflow isn't common, but can be useful in very big projects, or in highly hierarchical environments. It allows the project leader (the dictator) to delegate much of the work and collect large subsets of code at multiple points before integrating them.

Workflows Summary

These are some commonly used workflows that are possible with a distributed system like Git, but you can see that many variations are possible to suit your particular real-world workflow. Now that you can (hopefully) determine which workflow combination may work for you, we'll cover some more specific exam-

ples of how to accomplish the main roles that make up the different flows. In the next section, you'll learn about a few common patterns for contributing to a project.

Contributing to a Project

The main difficulty with describing how to contribute to a project is that there are a huge number of variations on how it's done. Because Git is very flexible, people can and do work together in many ways, and it's problematic to describe how you should contribute – every project is a bit different. Some of the variables involved are active contributor count, chosen workflow, your commit access, and possibly the external contribution method.

The first variable is active contributor count – how many users are actively contributing code to this project, and how often? In many instances, you'll have two or three developers with a few commits a day, or possibly less for somewhat dormant projects. For larger companies or projects, the number of developers could be in the thousands, with hundreds or thousands of commits coming in each day. This is important because with more and more developers, you run into more issues with making sure your code applies cleanly or can be easily merged. Changes you submit may be rendered obsolete or severely broken by work that is merged in while you were working or while your changes were waiting to be approved or applied. How can you keep your code consistently up to date and your commits valid?

The next variable is the workflow in use for the project. Is it centralized, with each developer having equal write access to the main codeline? Does the project have a maintainer or integration manager who checks all the patches? Are all the patches peer-reviewed and approved? Are you involved in that process? Is a lieutenant system in place, and do you have to submit your work to them first?

The next issue is your commit access. The workflow required in order to contribute to a project is much different if you have write access to the project than if you don't. If you don't have write access, how does the project prefer to accept contributed work? Does it even have a policy? How much work are you contributing at a time? How often do you contribute?

All these questions can affect how you contribute effectively to a project and what workflows are preferred or available to you. We'll cover aspects of each of these in a series of use cases, moving from simple to more complex; you should be able to construct the specific workflows you need in practice from these examples.

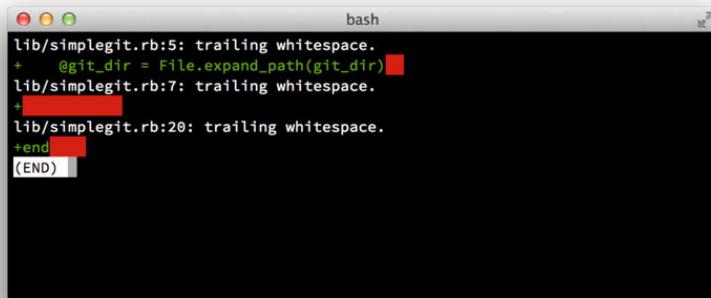
Commit Guidelines

Before we start looking at the specific use cases, here's a quick note about commit messages. Having a good guideline for creating commits and sticking to it makes working with Git and collaborating with others a lot easier. The Git project provides a document that lays out a number of good tips for creating commits from which to submit patches – you can read it in the Git source code in the [Documentation/SubmittingPatches](#) file.

First, you don't want to submit any whitespace errors. Git provides an easy way to check for this – before you commit, run `git diff --check`, which identifies possible whitespace errors and lists them for you.

FIGURE 5-4

Output of `git diff --check`.



```
bash
lib/simplegit.rb:5: trailing whitespace.
+ @git_dir = File.expand_path(git_dir)
lib/simplegit.rb:7: trailing whitespace.
+
lib/simplegit.rb:20: trailing whitespace.
+end
(END)
```

If you run that command before committing, you can tell if you're about to commit whitespace issues that may annoy other developers.

Next, try to make each commit a logically separate changeset. If you can, try to make your changes digestible – don't code for a whole weekend on five different issues and then submit them all as one massive commit on Monday. Even if you don't commit during the weekend, use the staging area on Monday to split your work into at least one commit per issue, with a useful message per commit. If some of the changes modify the same file, try to use `git add --patch` to partially stage files (covered in detail in "[Interactive Staging](#)"). The project snapshot at the tip of the branch is identical whether you do one commit or five, as long as all the changes are added at some point, so try to make things easier on your fellow developers when they have to review your changes. This approach also makes it easier to pull out or revert one of the changesets if you need to later. "[Rewriting History](#)" describes a number of useful Git tricks

for rewriting history and interactively staging files – use these tools to help craft a clean and understandable history before sending the work to someone else.

The last thing to keep in mind is the commit message. Getting in the habit of creating quality commit messages makes using and collaborating with Git a lot easier. As a general rule, your messages should start with a single line that's no more than about 50 characters and that describes the changeset concisely, followed by a blank line, followed by a more detailed explanation. The Git project requires that the more detailed explanation include your motivation for the change and contrast its implementation with previous behavior – this is a good guideline to follow. It's also a good idea to use the imperative present tense in these messages. In other words, use commands. Instead of “I added tests for” or “Adding tests for,” use “Add tests for.” Here is a template originally written by Tim Pope:

Short (50 chars or less) summary of changes

More detailed explanatory text, if necessary. Wrap it to about 72 characters or so. In some contexts, the first line is treated as the subject of an email and the rest of the text as the body. The blank line separating the summary from the body is critical (unless you omit the body entirely); tools like rebase can get confused if you run the two together.

Further paragraphs come after blank lines.

- Bullet points are okay, too
- Typically a hyphen or asterisk is used for the bullet, preceded by a single space, with blank lines in between, but conventions vary here

If all your commit messages look like this, things will be a lot easier for you and the developers you work with. The Git project has well-formatted commit messages – try running `git log --no-merges` there to see what a nicely formatted project-commit history looks like.

In the following examples, and throughout most of this book, for the sake of brevity this book doesn't have nicely-formatted messages like this; instead, we use the `-m` option to `git commit`. Do as we say, not as we do.

Private Small Team

The simplest setup you're likely to encounter is a private project with one or two other developers. “Private,” in this context, means closed-source – not ac-

cessible to the outside world. You and the other developers all have push access to the repository.

In this environment, you can follow a workflow similar to what you might do when using Subversion or another centralized system. You still get the advantages of things like offline committing and vastly simpler branching and merging, but the workflow can be very similar; the main difference is that merges happen client-side rather than on the server at commit time. Let's see what it might look like when two developers start to work together with a shared repository. The first developer, John, clones the repository, makes a change, and commits locally. (The protocol messages have been replaced with ... in these examples to shorten them somewhat.)

```
# John's Machine
$ git clone john@githost:simplegit.git
Initialized empty Git repository in /home/john/simplegit/.git/
...
$ cd simplegit/
$ vim lib/simplegit.rb
$ git commit -am 'removed invalid default value'
[master 738ee87] removed invalid default value
 1 files changed, 1 insertions(+), 1 deletions(-)
```

The second developer, Jessica, does the same thing – clones the repository and commits a change:

```
# Jessica's Machine
$ git clone jessica@githost:simplegit.git
Initialized empty Git repository in /home/jessica/simplegit/.git/
...
$ cd simplegit/
$ vim TODO
$ git commit -am 'add reset task'
[master fbff5bc] add reset task
 1 files changed, 1 insertions(+), 0 deletions(-)
```

Now, Jessica pushes her work up to the server:

```
# Jessica's Machine
$ git push origin master
...
To jessica@githost:simplegit.git
 1edee6b..fbff5bc  master -> master
```

John tries to push his change up, too:

```
# John's Machine
$ git push origin master
To john@githost:simplegit.git
 ! [rejected]      master -> master (non-fast forward)
error: failed to push some refs to 'john@githost:simplegit.git'
```

John isn't allowed to push because Jessica has pushed in the meantime. This is especially important to understand if you're used to Subversion, because you'll notice that the two developers didn't edit the same file. Although Subversion automatically does such a merge on the server if different files are edited, in Git you must merge the commits locally. John has to fetch Jessica's changes and merge them in before he will be allowed to push:

```
$ git fetch origin
...
From john@githost:simplegit
 + 049d078...fbff5bc master      -> origin/master
```

At this point, John's local repository looks something like this:

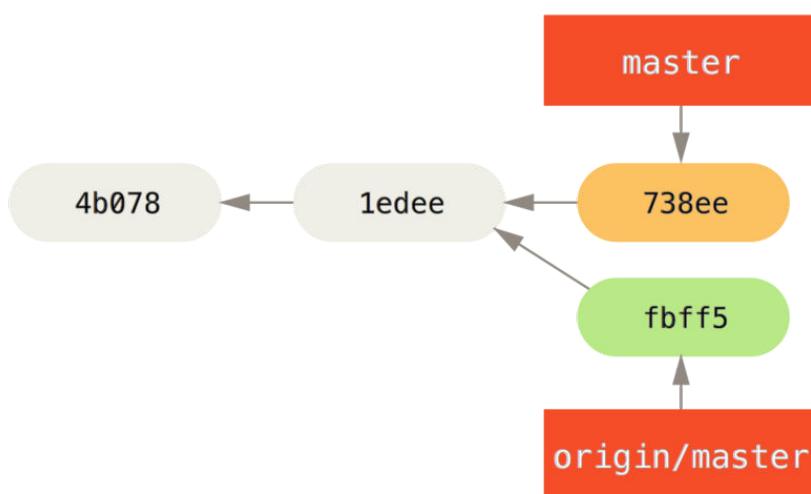


FIGURE 5-5
John's divergent history.

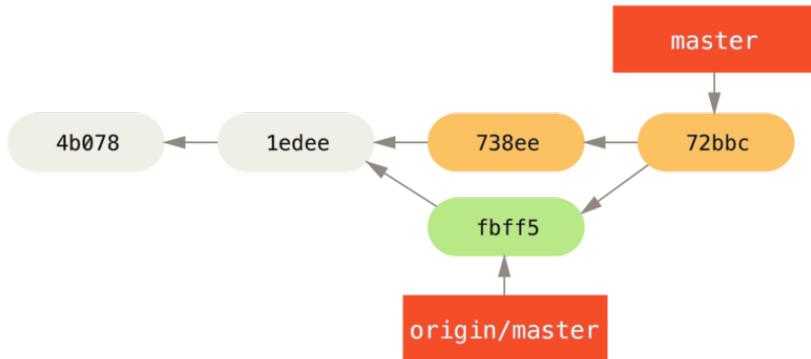
John has a reference to the changes Jessica pushed up, but he has to merge them into his own work before he is allowed to push:

```
$ git merge origin/master
Merge made by recursive.
 TODO | 1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

The merge goes smoothly – John’s commit history now looks like this:

FIGURE 5-6

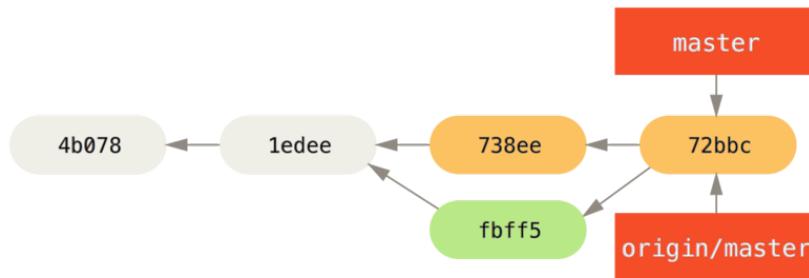
John’s repository after merging origin/master.



Now, John can test his code to make sure it still works properly, and then he can push his new merged work up to the server:

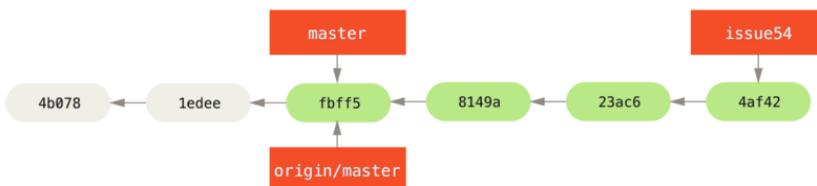
```
$ git push origin master
...
To john@githost:simplegit.git
 fbff5bc..72bbc59  master -> master
```

Finally, John’s commit history looks like this:

**FIGURE 5-7**

John’s history after pushing to the origin server.

In the meantime, Jessica has been working on a topic branch. She’s created a topic branch called `issue54` and done three commits on that branch. She hasn’t fetched John’s changes yet, so her commit history looks like this:

**FIGURE 5-8**

Jessica’s topic branch.

Jessica wants to sync up with John, so she fetches:

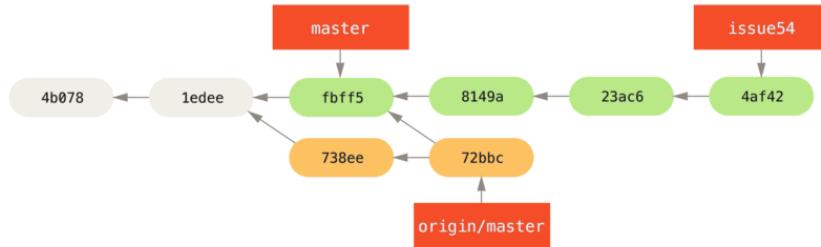
```

# Jessica's Machine
$ git fetch origin
...
From jessica@githost:simplegit
fbff5bc..72bbc59  master      -> origin/master
  
```

That pulls down the work John has pushed up in the meantime. Jessica’s history now looks like this:

FIGURE 5-9

Jessica's history after fetching John's changes.



Jessica thinks her topic branch is ready, but she wants to know what she has to merge into her work so that she can push. She runs `git log` to find out:

```
$ git log --no-merges issue54..origin/master
commit 738ee872852dfa9d6634e0dea7a324040193016
Author: John Smith <jsmith@example.com>
Date:   Fri May 29 16:01:27 2009 -0700

        removed invalid default value
```

The `issue54..origin/master` syntax is a log filter that asks Git to only show the list of commits that are on the latter branch (in this case `origin/master`) that are not on the first branch (in this case `issue54`). We'll go over this syntax in detail in “**Commit Ranges**”.

For now, we can see from the output that there is a single commit that John has made that Jessica has not merged in. If she merges `origin/master`, that is the single commit that will modify her local work.

Now, Jessica can merge her topic work into her `master` branch, merge John’s work (`origin/master`) into her `master` branch, and then push back to the server again. First, she switches back to her `master` branch to integrate all this work:

```
$ git checkout master
Switched to branch 'master'
Your branch is behind 'origin/master' by 2 commits, and can be fast-forwarded.
```

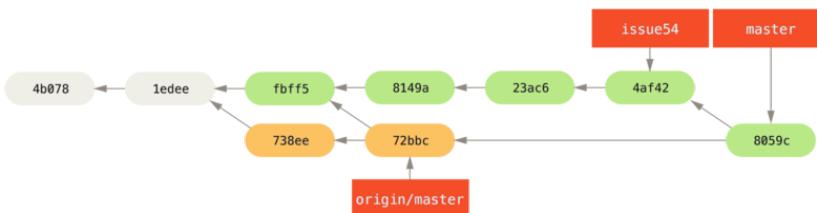
She can merge either `origin/master` or `issue54` first – they’re both upstream, so the order doesn’t matter. The end snapshot should be identical no matter which order she chooses; only the history will be slightly different. She chooses to merge in `issue54` first:

```
$ git merge issue54
Updating fbff5bc..4af4298
Fast forward
 README           |    1 +
 lib/simplegit.rb |   6 +++++-
 2 files changed, 6 insertions(+), 1 deletions(-)
```

No problems occur; as you can see it was a simple fast-forward. Now Jessica merges in John's work (`origin/master`):

```
$ git merge origin/master
Auto-merging lib/simplegit.rb
Merge made by recursive.
 lib/simplegit.rb |    2 ++
 1 files changed, 2 insertions(+), 0 deletions(-)
```

Everything merges cleanly, and Jessica's history looks like this:

**FIGURE 5-10**

Jessica's history after merging John's changes.

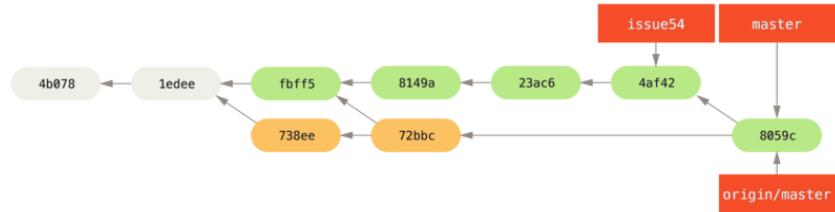
Now `origin/master` is reachable from Jessica's `master` branch, so she should be able to successfully push (assuming John hasn't pushed again in the meantime):

```
$ git push origin master
...
To jessica@githost:simplegit.git
 72bbc59..8059c15  master -> master
```

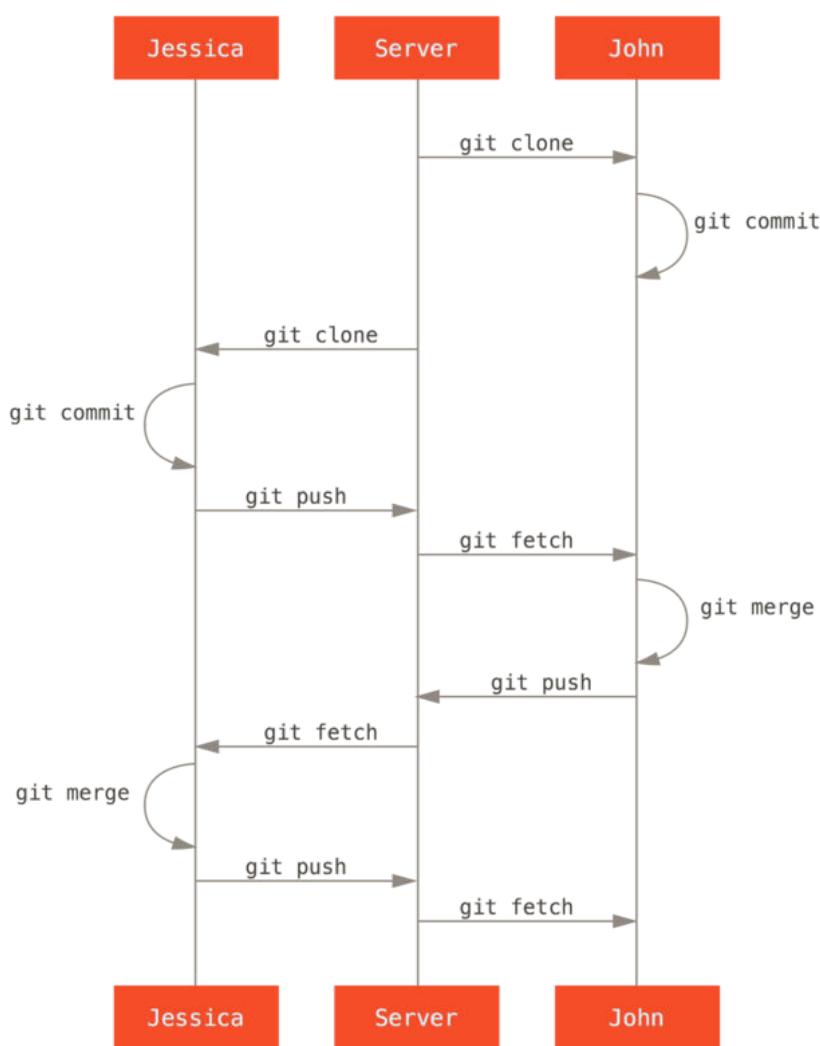
Each developer has committed a few times and merged each other's work successfully.

FIGURE 5-11

*Jessica's history
after pushing all
changes back to the
server.*



That is one of the simplest workflows. You work for a while, generally in a topic branch, and merge into your master branch when it's ready to be integrated. When you want to share that work, you merge it into your own master branch, then fetch and merge `origin/master` if it has changed, and finally push to the `master` branch on the server. The general sequence is something like this:

**FIGURE 5-12**

General sequence of events for a simple multiple-developer Git workflow.

Private Managed Team

In this next scenario, you'll look at contributor roles in a larger private group. You'll learn how to work in an environment where small groups collaborate on features and then those team-based contributions are integrated by another party.

Let's say that John and Jessica are working together on one feature, while Jessica and Josie are working on a second. In this case, the company is using a

type of integration-manager workflow where the work of the individual groups is integrated only by certain engineers, and the `master` branch of the main repo can be updated only by those engineers. In this scenario, all work is done in team-based branches and pulled together by the integrators later.

Let's follow Jessica's workflow as she works on her two features, collaborating in parallel with two different developers in this environment. Assuming she already has her repository cloned, she decides to work on `featureA` first. She creates a new branch for the feature and does some work on it there:

```
# Jessica's Machine
$ git checkout -b featureA
Switched to a new branch 'featureA'
$ vim lib/simplegit.rb
$ git commit -am 'add limit to log function'
[featureA 3300904] add limit to log function
 1 files changed, 1 insertions(+), 1 deletions(-)
```

At this point, she needs to share her work with John, so she pushes her `featureA` branch commits up to the server. Jessica doesn't have push access to the `master` branch – only the integrators do – so she has to push to another branch in order to collaborate with John:

```
$ git push -u origin featureA
...
To jessica@githost:simplegit.git
 * [new branch]      featureA -> featureA
```

Jessica e-mails John to tell him that she's pushed some work into a branch named `featureA` and he can look at it now. While she waits for feedback from John, Jessica decides to start working on `featureB` with Josie. To begin, she starts a new feature branch, basing it off the server's `master` branch:

```
# Jessica's Machine
$ git fetch origin
$ git checkout -b featureB origin/master
Switched to a new branch 'featureB'
```

Now, Jessica makes a couple of commits on the `featureB` branch:

```
$ vim lib/simplegit.rb
$ git commit -am 'made the ls-tree function recursive'
```

```
[featureB e5b0fdc] made the ls-tree function recursive
 1 files changed, 1 insertions(+), 1 deletions(-)
$ vim lib/simplegit.rb
$ git commit -am 'add ls-files'
[featureB 8512791] add ls-files
 1 files changed, 5 insertions(+), 0 deletions(-)
```

Jessica's repository looks like this:

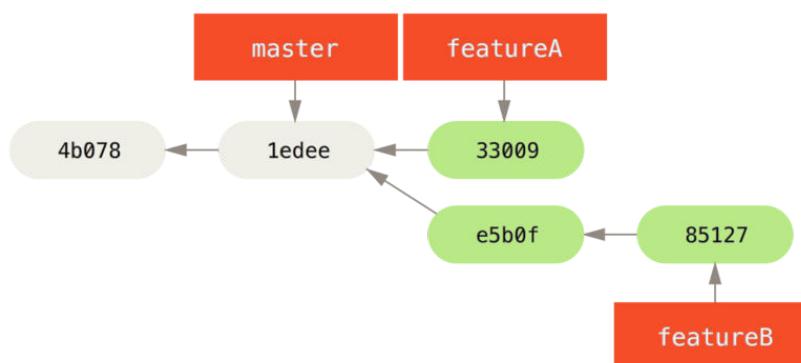


FIGURE 5-13

Jessica's initial commit history.

She's ready to push up her work, but gets an e-mail from Josie that a branch with some initial work on it was already pushed to the server as `featureBee`. Jessica first needs to merge those changes in with her own before she can push to the server. She can then fetch Josie's changes down with `git fetch`:

```
$ git fetch origin
...
From jessica@githost:simplegit
 * [new branch]      featureBee -> origin/featureBee
```

Jessica can now merge this into the work she did with `git merge`:

```
$ git merge origin/featureBee
Auto-merging lib/simplegit.rb
Merge made by recursive.
lib/simplegit.rb |    4 +---
1 files changed, 4 insertions(+), 0 deletions(-)
```

There is a bit of a problem – she needs to push the merged work in her `featureB` branch to the `featureBee` branch on the server. She can do so by specifying the local branch followed by a colon (:) followed by the remote branch to the `git push` command:

```
$ git push -u origin featureB:featureBee
...
To jessica@githost:simplegit.git
  fba9af8..cd685d1  featureB -> featureBee
```

This is called a *refspec*. See “[The Refspec](#)” for a more detailed discussion of Git refspecs and different things you can do with them. Also notice the `-u` flag; this is short for `--set-upstream`, which configures the branches for easier pushing and pulling later.

Next, John e-mails Jessica to say he’s pushed some changes to the `featureA` branch and asks her to verify them. She runs a `git fetch` to pull down those changes:

```
$ git fetch origin
...
From jessica@githost:simplegit
  3300904..aad881d  featureA -> origin/featureA
```

Then, she can see what has been changed with `git log`:

```
$ git log featureA..origin/featureA
commit aad881d154acdaeb2b6b18ea0e827ed8a6d671e6
Author: John Smith <jsmith@example.com>
Date:   Fri May 29 19:57:33 2009 -0700

        changed log output to 30 from 25
```

Finally, she merges John’s work into her own `featureA` branch:

```
$ git checkout featureA
Switched to branch 'featureA'
$ git merge origin/featureA
Updating 3300904..aad881d
Fast forward
  lib/simplegit.rb | 10 ++++++----
  1 files changed, 9 insertions(+), 1 deletions(-)
```

Jessica wants to tweak something, so she commits again and then pushes this back up to the server:

```
$ git commit -am 'small tweak'
[featureA 774b3ed] small tweak
 1 files changed, 1 insertions(+), 1 deletions(-)
$ git push
...
To jessica@githost:simplegit.git
 3300904..774b3ed  featureA -> featureA
```

Jessica's commit history now looks something like this:

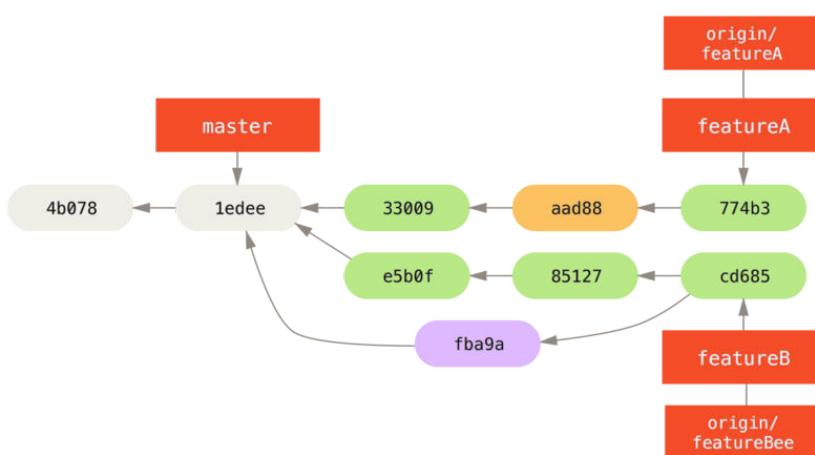
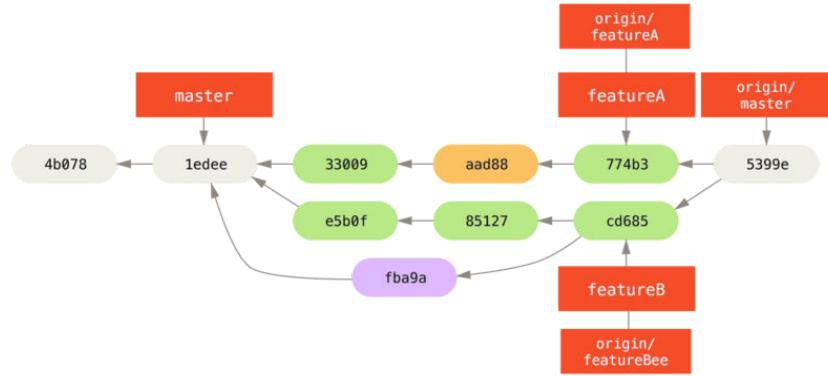


FIGURE 5-14
Jessica's history
after committing on
a feature branch.

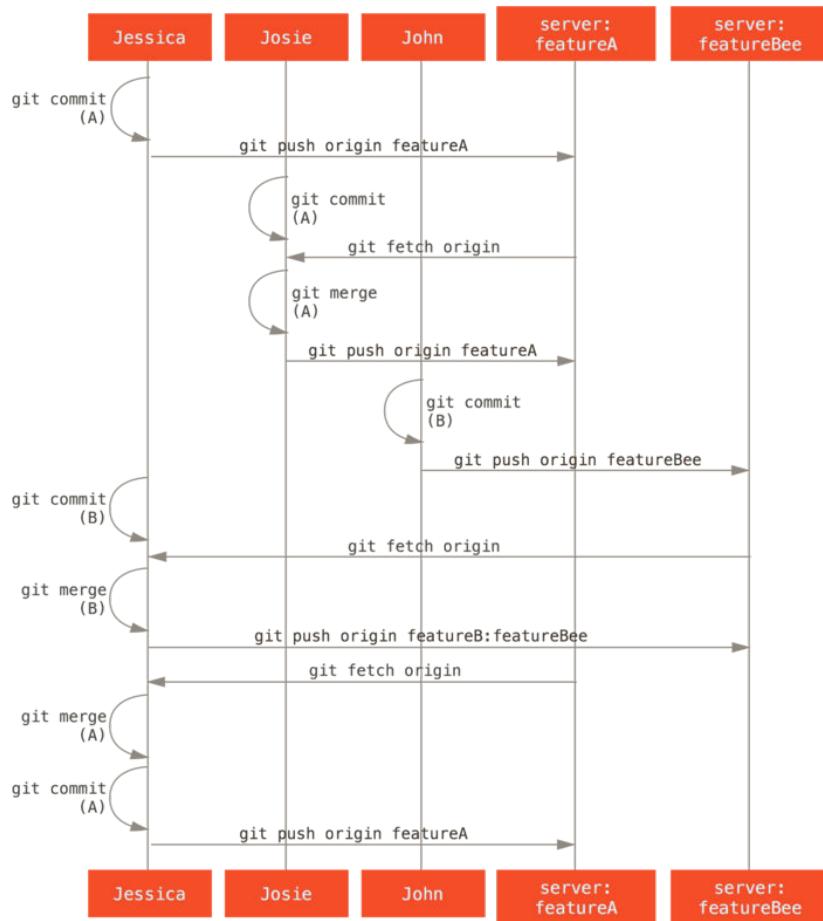
Jessica, Josie, and John inform the integrators that the `featureA` and `featureBee` branches on the server are ready for integration into the mainline. After the integrators merge these branches into the mainline, a fetch will bring down the new merge commit, making the history look like this:

FIGURE 5-15

Jessica's history after merging both her topic branches.



Many groups switch to Git because of this ability to have multiple teams working in parallel, merging the different lines of work late in the process. The ability of smaller subgroups of a team to collaborate via remote branches without necessarily having to involve or impede the entire team is a huge benefit of Git. The sequence for the workflow you saw here is something like this:

**FIGURE 5-16**

Basic sequence of this managed-team workflow.

Forked Public Project

Contributing to public projects is a bit different. Because you don't have the permissions to directly update branches on the project, you have to get the work to the maintainers some other way. This first example describes contributing via forking on Git hosts that support easy forking. Many hosting sites support this (including GitHub, BitBucket, Google Code, repo.or.cz, and others), and many project maintainers expect this style of contribution. The next section deals with projects that prefer to accept contributed patches via e-mail.

First, you'll probably want to clone the main repository, create a topic branch for the patch or patch series you're planning to contribute, and do your work there. The sequence looks basically like this:

```
$ git clone (url)
$ cd project
$ git checkout -b featureA
# (work)
$ git commit
# (work)
$ git commit
```

You may want to use `rebase -i` to squash your work down to a single commit, or rearrange the work in the commits to make the patch easier for the maintainer to review – see “[Rewriting History](#)” for more information about interactive rebasing.

When your branch work is finished and you’re ready to contribute it back to the maintainers, go to the original project page and click the “Fork” button, creating your own writable fork of the project. You then need to add in this new repository URL as a second remote, in this case named `myfork`:

```
$ git remote add myfork (url)
```

Then you need to push your work up to it. It’s easiest to push the topic branch you’re working on up to your repository, rather than merging into your master branch and pushing that up. The reason is that if the work isn’t accepted or is cherry picked, you don’t have to rewind your master branch. If the maintainers merge, rebase, or cherry-pick your work, you’ll eventually get it back via pulling from their repository anyhow:

```
$ git push -u myfork featureA
```

When your work has been pushed up to your fork, you need to notify the maintainer. This is often called a pull request, and you can either generate it via the website – GitHub has its own Pull Request mechanism that we’ll go over in [Chapter 6](#) – or you can run the `git request-pull` command and e-mail the output to the project maintainer manually.

The `request-pull` command takes the base branch into which you want your topic branch pulled and the Git repository URL you want them to pull from, and outputs a summary of all the changes you’re asking to be pulled in. For instance, if Jessica wants to send John a pull request, and she’s done two commits on the topic branch she just pushed up, she can run this:

```
$ git request-pull origin/master myfork
The following changes since commit 1edee6b1d61823a2de3b09c160d7080b8d1b3a40:
  John Smith (1):
    added a new function

are available in the git repository at:

  git://githost/simplegit.git featureA

  Jessica Smith (2):
    add limit to log function
    change log output to 30 from 25

  lib/simplegit.rb |   10 ++++++++-+
  1 files changed, 9 insertions(+), 1 deletions(-)
```

The output can be sent to the maintainer – it tells them where the work was branched from, summarizes the commits, and tells where to pull this work from.

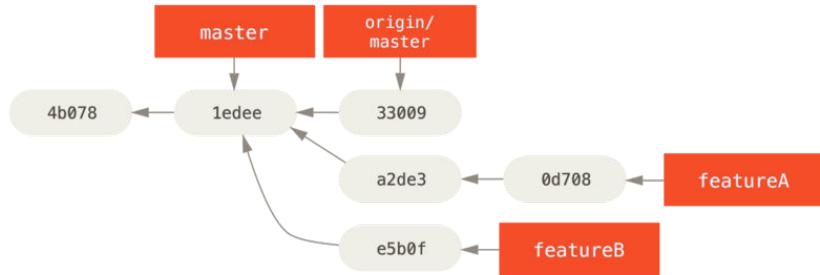
On a project for which you’re not the maintainer, it’s generally easier to have a branch like `master` always track `origin/master` and to do your work in topic branches that you can easily discard if they’re rejected. Having work themes isolated into topic branches also makes it easier for you to rebase your work if the tip of the main repository has moved in the meantime and your commits no longer apply cleanly. For example, if you want to submit a second topic of work to the project, don’t continue working on the topic branch you just pushed up – start over from the main repository’s `master` branch:

```
$ git checkout -b featureB origin/master
# (work)
$ git commit
$ git push myfork featureB
# (email maintainer)
$ git fetch origin
```

Now, each of your topics is contained within a silo – similar to a patch queue – that you can rewrite, rebase, and modify without the topics interfering or interdepending on each other, like so:

FIGURE 5-17

Initial commit history with featureB work.



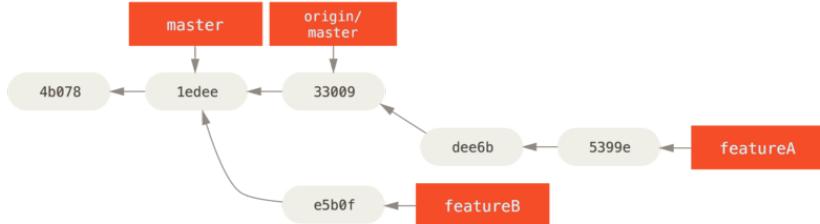
Let's say the project maintainer has pulled in a bunch of other patches and tried your first branch, but it no longer cleanly merges. In this case, you can try to rebase that branch on top of `origin/master`, resolve the conflicts for the maintainer, and then resubmit your changes:

```
$ git checkout featureA
$ git rebase origin/master
$ git push -f myfork featureA
```

This rewrites your history to now look like **Figure 5-18**.

FIGURE 5-18

Commit history after featureA work.



Because you rebased the branch, you have to specify the `-f` to your push command in order to be able to replace the `featureA` branch on the server with a commit that isn't a descendant of it. An alternative would be to push this new work to a different branch on the server (perhaps called `featureAv2`).

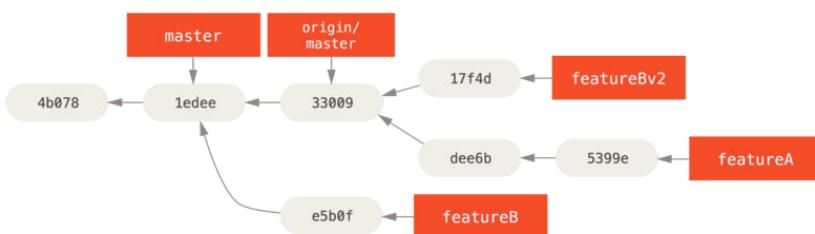
Let's look at one more possible scenario: the maintainer has looked at work in your second branch and likes the concept but would like you to change an implementation detail. You'll also take this opportunity to move the work to be

based off the project's current `master` branch. You start a new branch based off the current `origin/master` branch, squash the `featureB` changes there, resolve any conflicts, make the implementation change, and then push that up as a new branch:

```
$ git checkout -b featureBv2 origin/master
$ git merge --no-commit --squash featureB
# (change implementation)
$ git commit
$ git push myfork featureBv2
```

The `--squash` option takes all the work on the merged branch and squashes it into one non-merge commit on top of the branch you're on. The `--no-commit` option tells Git not to automatically record a commit. This allows you to introduce all the changes from another branch and then make more changes before recording the new commit.

Now you can send the maintainer a message that you've made the requested changes and they can find those changes in your `featureBv2` branch.

**FIGURE 5-19**

Commit history after featureBv2 work.

Public Project over E-Mail

Many projects have established procedures for accepting patches – you'll need to check the specific rules for each project, because they will differ. Since there are several older, larger projects which accept patches via a developer mailing list, we'll go over an example of that now.

The workflow is similar to the previous use case – you create topic branches for each patch series you work on. The difference is how you submit them to the project. Instead of forking the project and pushing to your own writable version, you generate e-mail versions of each commit series and e-mail them to the developer mailing list:

```
$ git checkout -b topicA
# (work)
$ git commit
# (work)
$ git commit
```

Now you have two commits that you want to send to the mailing list. You use `git format-patch` to generate the mbox-formatted files that you can e-mail to the list – it turns each commit into an e-mail message with the first line of the commit message as the subject and the rest of the message plus the patch that the commit introduces as the body. The nice thing about this is that applying a patch from an e-mail generated with `format-patch` preserves all the commit information properly.

```
$ git format-patch -M origin/master
0001-add-limit-to-log-function.patch
0002-changed-log-output-to-30-from-25.patch
```

The `format-patch` command prints out the names of the patch files it creates. The `-M` switch tells Git to look for renames. The files end up looking like this:

```
$ cat 0001-add-limit-to-log-function.patch
From 330090432754092d704da8e76ca5c05c198e71a8 Mon Sep 17 00:00:00 2001
From: Jessica Smith <jessica@example.com>
Date: Sun, 6 Apr 2008 10:17:23 -0700
Subject: [PATCH 1/2] add limit to log function

Limit log functionality to the first 20

---
lib/simplegit.rb |    2 ++
1 files changed, 1 insertions(+), 1 deletions(-)

diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index 76f47bc..f9815f1 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -14,7 +14,7 @@ class SimpleGit
end

def log(treeish = 'master')
-   command("git log #{treeish}")
+   command("git log -n 20 #{treeish}")
```

```
end

def ls_tree(treeish = 'master')
-- 
2.1.0
```

You can also edit these patch files to add more information for the e-mail list that you don't want to show up in the commit message. If you add text between the `--` line and the beginning of the patch (the `diff --git` line), then developers can read it; but applying the patch excludes it.

To e-mail this to a mailing list, you can either paste the file into your e-mail program or send it via a command-line program. Pasting the text often causes formatting issues, especially with “smarter” clients that don't preserve newlines and other whitespace appropriately. Luckily, Git provides a tool to help you send properly formatted patches via IMAP, which may be easier for you. We'll demonstrate how to send a patch via Gmail, which happens to be the e-mail agent we know best; you can read detailed instructions for a number of mail programs at the end of the aforementioned Documentation/SubmittingPatches file in the Git source code.

First, you need to set up the `imap` section in your `~/.gitconfig` file. You can set each value separately with a series of `git config` commands, or you can add them manually, but in the end your config file should look something like this:

```
[imap]
folder = "[Gmail]/Drafts"
host = imaps://imap.gmail.com
user = user@gmail.com
pass = p4ssw0rd
port = 993
sslverify = false
```

If your IMAP server doesn't use SSL, the last two lines probably aren't necessary, and the host value will be `imap://` instead of `imaps://`. When that is set up, you can use `git send-email` to place the patch series in the Drafts folder of the specified IMAP server:

```
$ git send-email *.patch
0001-added-limit-to-log-function.patch
0002-changed-log-output-to-30-from-25.patch
Who should the emails appear to be from? [Jessica Smith <jessica@example.com>]
Emails will be sent from: Jessica Smith <jessica@example.com>
```

```
Who should the emails be sent to? jessica@example.com  
Message-ID to be used as In-Reply-To for the first email? y
```

Then, Git spits out a bunch of log information looking something like this for each patch you're sending:

```
(mbox) Adding cc: Jessica Smith <jessica@example.com> from  
      \line 'From: Jessica Smith <jessica@example.com>'  
OK. Log says:  
Sendmail: /usr/sbin/sendmail -i jessica@example.com  
From: Jessica Smith <jessica@example.com>  
To: jessica@example.com  
Subject: [PATCH 1/2] added limit to log function  
Date: Sat, 30 May 2009 13:29:15 -0700  
Message-Id: <1243715356-61726-1-git-send-email-jessica@example.com>  
X-Mailer: git-send-email 1.6.2.rc1.20.g8c5b.dirty  
In-Reply-To: <y>  
References: <y>
```

Result: OK

At this point, you should be able to go to your Drafts folder, change the To field to the mailing list you're sending the patch to, possibly CC the maintainer or person responsible for that section, and send it off.

Summary

This section has covered a number of common workflows for dealing with several very different types of Git projects you're likely to encounter, and introduced a couple of new tools to help you manage this process. Next, you'll see how to work the other side of the coin: maintaining a Git project. You'll learn how to be a benevolent dictator or integration manager.

Manteniendo un proyecto

Además de saber cómo contribuir de manera efectiva a un proyecto, probablemente necesitarás saber cómo mantenerlo. Esto puede comprender desde aceptar y aplicar parches generados vía `format-patch` y enviados por e-mail, hasta integrar cambios en ramas remotas para repositorios que has añadido como remotos a tu proyecto. Tanto si mantienes un repositorio canónico como si quieres ayudar verificando o aprobando parches, necesitas conocer cómo aceptar trabajo de otros colaboradores de la forma lo más clara y sostenible posible a largo plazo.

Trabajando en ramas puntuales

Cuando estás pensando en integrar nuevo trabajo, generalmente es una buena idea probarlo en una rama puntual (topic branch) - una rama temporal específicamente creada para probar ese nuevo trabajo. De esta forma, es fácil ajustar un parche individualmente y abandonarlo si no funciona hasta que tengas tiempo de retomarlo. Si creas una rama simple con un nombre relacionado con el trabajo que vas a probar, como `ruby_client` o algo igualmente descriptivo, puedes recordarlo fácilmente si tienes que abandonarlo y retomarlo posteriormente. El responsable del mantenimiento del proyecto Git también tiende a usar una nomenclatura con estas ramas – como `sc/ruby_client`, donde `sc` es la abreviatura de la persona que envió el trabajo. Como recordarás, puedes crear la rama a partir de la rama master de la siguiente forma:

```
$ git branch sc/ruby_client master
```

O, si quieres cambiar inmediatamente a la rama, puedes usar la opción `checkout -b`:

```
$ git checkout -b sc/ruby_client master
```

Ahora estás listo para añadir el trabajo recibido en esta rama puntual y decidir si quieres incorporarlo en tus ramas de largo recorrido.

Aplicando parches recibidos por e-mail

Si recibes por e-mail un parche que necesitas integrar en tu proyecto, deberías aplicarlo en tu rama puntual para evaluarlo. Hay dos formas de aplicar un parche enviado por e-mail: con `git apply` o `git am`.

APLICANDO UN PARCHE CON APPLY

Si recibiste el parche de alguien que lo generó con `git diff` o con el comando Unix `diff` (lo cual no se recomienda; consulta la siguiente sección), puedes aplicarlo con el comando `git apply`. Suponiendo que guardaste el parche en `/tmp/patch-ruby-client.patch`, puedes aplicarlo de esta forma:

```
$ git apply /tmp/patch-ruby-client.patch
```

Esto modifica los ficheros en tu directorio de trabajo. Es casi idéntico a ejecutar un comando `patch -p1` para aplicar el parche, aunque es más paranoico y acepta menos coincidencias aproximadas que `patch`. También puede manejar ficheros nuevos, borrados y renombrados si están descritos en formato `git diff`, mientras que `patch` no puede hacerlo. Por último, `git apply` sigue un modelo “aplica todo o aborta todo”, donde se aplica todo o nada, mientras que `patch` puede aplicar parches parcialmente, dejando tu directorio de trabajo en un estado inconsistente. `git apply` es en general mucho más conservador que `patch`. No creará un `commit` por ti – tras ejecutarlo, debes preparar (`stage`) y confirmar (`commit`) manualmente los cambios introducidos.

También puedes usar `git apply` para comprobar si un parche se aplica de forma limpia antes de aplicarlo realmente – puedes ejecutar `git apply --check` indicando el parche:

```
$ git apply --check 0001-seeing-if-this-helps-the-gem.patch
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
```

Si no obtienes salida, entonces el parche debería aplicarse limpiamente. Este comando también devuelve un estado distinto de cero si la comprobación falla, por lo que puedes usarlo en scripts.

APLICANDO UN PARCHE CON AM

Si el colaborador es usuario de Git y conoce lo suficiente como para usar el comando `format-patch` para generar el parche, entonces tu trabajo es más sencillo, ya que el parche ya contiene información del autor y un mensaje de `commit`. Si puedes, anima a tus colaboradores a usar `format-patch` en lugar de `diff` para generar parches. Sólo deberías usar `git apply` para parches anti-guos y cosas similares.

Para aplicar un parche generado con `format-patch`, usa `git am`. Técnicamente, `git am` se construyó para leer de un fichero `mbox` (buzón de correo). Es un formato de texto plano simple para almacenar uno o más mensajes de correo en un fichero de texto. Es algo parecido a esto:

```
From 330090432754092d704da8e76ca5c05c198e71a8 Mon Sep 17 00:00:00 2001
From: Jessica Smith <jessica@example.com>
Date: Sun, 6 Apr 2008 10:17:23 -0700
Subject: [PATCH 1/2] add limit to log function
```

```
Limit log functionality to the first 20
```

Esto es el comienzo de la salida del comando `format-patch` que viste en la sección anterior. También es un formato mbox válido. Si alguien te ha enviado el parche por e-mail usando `git send-email` y lo has descargado en formato mbox, entonces puedes pasar ese fichero a `git am` y comenzará a aplicar todos los parches que encuentre. Si usas un cliente de correo que puede guardar varios e-mails en formato mbox, podrías guardar conjuntos completos de parches en un único fichero y a continuación usar `git am` para aplicarlos de uno en uno.

Sin embargo, si alguien subió a un sistema de gestión de incidencias o algo parecido un parche generado con `format-patch`, podrías guardar localmente el fichero y posteriormente pasarlo a `git am` para aplicarlo:

```
$ git am 0001-limit-log-function.patch
Applying: add limit to log function
```

Puedes ver que aplicó el parche limpiamente y creó automáticamente un nuevo commit. La información del autor se toma de las cabeceras `From` y `Date` del e-mail, y el mensaje del commit sale del `Subject` y el cuerpo del e-mail (antes del parche). Por ejemplo, si se aplicó este parche desde el fichero mbox del ejemplo anterior, el commit generado sería algo como esto:

```
$ git log --pretty=fuller -1
commit 6c5e70b984a60b3cecd395edd5b48a7575bf58e0
Author: Jessica Smith <jessica@example.com>
AuthorDate: Sun Apr 6 10:17:23 2008 -0700
Commit: Scott Chacon <schacon@gmail.com>
CommitDate: Thu Apr 9 09:19:06 2009 -0700

add limit to log function
```

```
Limit log functionality to the first 20
```

El campo `Commit` indica la persona que aplicó el parche y cuándo lo aplicó. El campo `Author` es la persona que creó originalmente el parche y cuándo fue creado.

Pero es posible que el parche no se aplique limpiamente. Quizás tu rama principal es muy diferente de la rama a partir de la cual se creó el parche, o el parche depende de otro parche que aún no has aplicado. En ese caso, el proceso `git am` fallará y te preguntará qué hacer:

```
$ git am 0001-seeing-if-this-helps-the-gem.patch
Applying: seeing if this helps the gem
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
Patch failed at 0001.
When you have resolved this problem run "git am --resolved".
If you would prefer to skip this patch, instead run "git am --skip".
To restore the original branch and stop patching run "git am --abort".
```

Este comando marca los conflictos en cualquier fichero para el cual detecte problemas, como si fuera una operación `merge` o `rebase`. Estos problemas se solucionan de la misma forma - edita el fichero para resolver el conflicto, prepara (stage) el nuevo fichero, y por último ejecuta `git am --resolved` para continuar con el siguiente parche:

```
$ (fix the file)
$ git add ticgit.gemspec
$ git am --resolved
Applying: seeing if this helps the gem
```

Si quieras que Git intente resolver el conflicto de forma un poco más inteligente, puedes indicar la opción `-3` para que Git intente hacer un merge a tres bandas. Esta opción no está activa por defecto, ya que no funciona si el commit en que el parche está basado no está en tu repositorio. Si tienes ese commit – si el parche partió de un commit público – entonces la opción `-3` es normalmente mucho más inteligente a la hora de aplicar un parche conflictivo:

```
$ git am -3 0001-seeing-if-this-helps-the-gem.patch
Applying: seeing if this helps the gem
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
Using index info to reconstruct a base tree...
Falling back to patching base and 3-way merge...
No changes -- Patch already applied.
```

En este caso, el parche ya ha sido aplicado. Sin la opción `-3`, parecería un conflicto.

Si estás aplicando varios parches a partir de un fichero `mbox`, también puedes ejecutar el comando `am` en modo interactivo, el cual se detiene en cada parche para preguntar si quieras aplicarlo:

```
$ git am -3 -i mbox
Commit Body is:
-----
seeing if this helps the gem
-----
Apply? [y]es/[n]o/[e]dit/[v]iew patch/[a]ccept all
```

Esto está bien si tienes guardados varios parches, ya que puedes revisar el parche previamente y no aplicarlo si ya lo has hecho.

Una vez aplicados y confirmados todos los parches de tu rama puntual, puedes decidir cómo y cuándo integrarlos en una rama de largo recorrido.

Recuperando ramas remotas

Si recibes una contribución de un usuario de Git que configuró su propio repositorio, realizó cambios en él y envió la URL del repositorio junto con el nombre de la rama remota donde están los cambios, puedes añadirlo como una rama remota y hacer integraciones (merges) de forma local.

Por ejemplo, si Jessica te envía un e-mail diciendo que tiene una nueva funcionalidad muy interesante en la rama `ruby-client` de su repositorio, puedes probarla añadiendo el repositorio remoto y recuperando localmente dicha rama:

```
$ git remote add jessica git://github.com/jessica/myproject.git
$ git fetch jessica
$ git checkout -b rubyclient jessica/ruby-client
```

Si más tarde te envía otro email con una nueva funcionalidad en otra rama, puedes recuperarla (fetch y check out) directamente porque ya tienes el repositorio remoto configurado.

Esto es más útil cuando trabajas regularmente con una persona. Sin embargo, si alguien sólo envía un parche de forma ocasional, aceptarlo vía email podría llevar menos tiempo que obligar a todo el mundo a ejecutar su propio servidor y tener que añadir y eliminar repositorios remotos continuamente para obtener unos cuantos parches. Además, probablemente no quieras tener cientos de repositorios remotos, uno por cada persona que envía uno o dos parches. En cualquier caso, los scripts y los servicios alojados pueden facilitar todo esto — depende en gran medida de cómo desarrollan tanto tus colaboradores como tú mismo —

Otra ventaja de esta opción es que además puedes obtener un historial de commits. Aunque pueden surgir los problemas habituales durante la integra-

ción (merge), al menos sabes en qué punto de tu historial se basa su trabajo; Por defecto, se realiza una integración a tres bandas, en lugar de indicar un -3 y esperar que el parche se generara a partir de un commit público al que tengas acceso.

Si no trabajas regularmente con alguien pero aún así quieras obtener sus contribuciones de esta manera, puedes pasar la URL del repositorio remoto al comando `git pull`. Esto recupera los cambios de forma puntual (pull) sin guardar la URL como una referencia remota:

```
$ git pull https://github.com/onetimeguy/project
From https://github.com/onetimeguy/project
 * branch           HEAD      -> FETCH_HEAD
Merge made by recursive.
```

Decidiendo qué introducir

Ahora tienes una rama puntual con trabajo de los colaboradores. En este punto, puedes decidir qué quieras hacer con ella. Esta sección repasa un par de mandos para que puedas ver cómo se usan para revisar exactamente qué se va a introducir si integras los cambios en tu rama principal.

A menudo es muy útil obtener una lista de todos los commits de una rama que no están en tu rama principal. Puedes excluir de dicha lista los commits de tu rama principal anteponiendo la opción `--not` al nombre de la rama. El efecto de esto es el mismo que el formato `master..contrib` que usamos anteriormente. Por ejemplo, si un colaborador te envía dos parches y creas una rama llamada `contrib` para aplicar los parches, puedes ejecutar esto:

```
$ git log contrib --not master
commit 5b6235bd297351589efc4d73316f0a68d484f118
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Oct 24 09:53:59 2008 -0700

        seeing if this helps the gem

commit 7482e0d16d04bea79d0dba8988cc78df655f16a0
Author: Scott Chacon <schacon@gmail.com>
Date:   Mon Oct 22 19:38:36 2008 -0700

        updated the gemspec to hopefully work better
```

Para ver qué cambios introduce cada commit, recuerda que puedes indicar la opción `-p` a `git log`, y añadirá las diferencias introducidas en cada commit.

Para tener una visión completa de qué ocurriría si integraras esta rama puntual en otra rama, podrías usar un sencillo truco para obtener los resultados correctos. Podrías pensar en ejecutar esto:

```
$ git diff master
```

Este comando te da las diferencias, pero los resultados podrían ser confusos. Si tu rama `master` ha avanzado desde que creaste la rama puntual, entonces obtendrás resultados aparentemente extraños. Esto ocurre porque Git compara directamente las instantáneas del último commit de la rama puntual en la que estás con la instantánea del último commit de la rama `master`. Por ejemplo, si has añadido una línea a un fichero en la rama `master`, al hacer una comparación directa de las instantáneas parecerá que la rama puntual va a eliminar esa línea.

Si `master` es un ancestro de tu rama puntual, esto no supone un problema; pero si los dos historiales divergen, al hacer una comparación directa parecerá que estás añadiendo todos los cambios nuevos en tu rama puntual y eliminándolos de la rama `master`.

Lo que realmente necesitas ver son los cambios añadidos en tu rama puntual – el trabajo que introducirás si integras esta rama en la `master`. Para conseguir esto, Git compara el último commit de tu rama puntual con el primer ancestro en común respecto a la rama `master`.

Técnicamente puedes hacer esto averiguando explícitamente el ancestro común y ejecutando el `diff` sobre dicho ancestro:

```
$ git merge-base contrib master
36c7dba2c95e6bbb78dfa822519ecfec6e1ca649
$ git diff 36c7db
```

Sin embargo, eso no es lo más conveniente, así que Git ofrece un atajo para hacer eso mismo: la sintaxis del triple-punto. En el contexto del comando `diff`, puedes poner tres puntos tras el nombre de una rama para hacer un `diff` entre el último commit de la rama en la que estás y su ancestro común con otra rama:

```
$ git diff master...contrib
```

Este comando sólo muestra el trabajo introducido en tu rama puntual actual desde su ancestro común con la rama `master`. Es una sintaxis muy útil a recordar.

Integrando el trabajo de los colaboradores

Cuando todo el trabajo de tu rama puntual está listo para ser integrado en una rama de largo recorrido, la cuestión es cómo hacerlo. Es más, ¿qué flujo de trabajo general quieres seguir para mantener el proyecto? Tienes varias opciones y vamos a ver algunas de ellas.

INTEGRANDO FLUJOS DE TRABAJO

Un flujo de trabajo sencillo integra tu trabajo en tu rama `master`. En este escenario, tienes una rama `master` que contiene básicamente código estable. Cuando tienes trabajo propio en una rama puntual o trabajo aportado por algún colaborador que ya has verificado, lo integras en tu rama `master`, borras la rama puntual y continúas el proceso. Si tenemos un repositorio con trabajo en dos ramas llamadas `ruby_client` y `php_client`, tal y como se muestra en **Figure 5-20** e integramos primero `ruby_client` y luego `php_client`, entonces tu historial terminará con este aspecto **Figure 5-21**.

FIGURE 5-20

Historial con varias ramas puntuales.

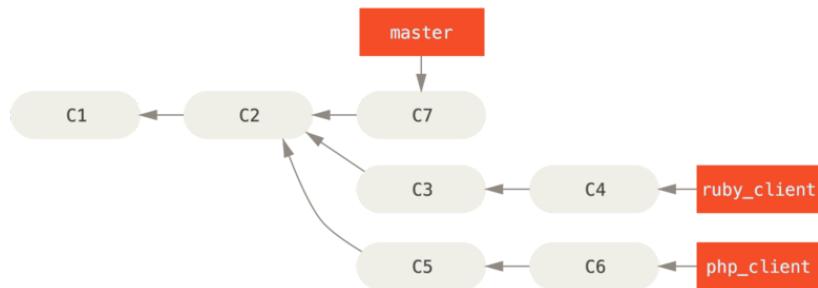
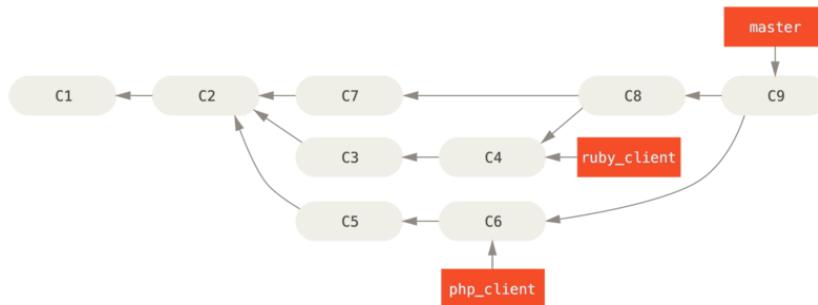


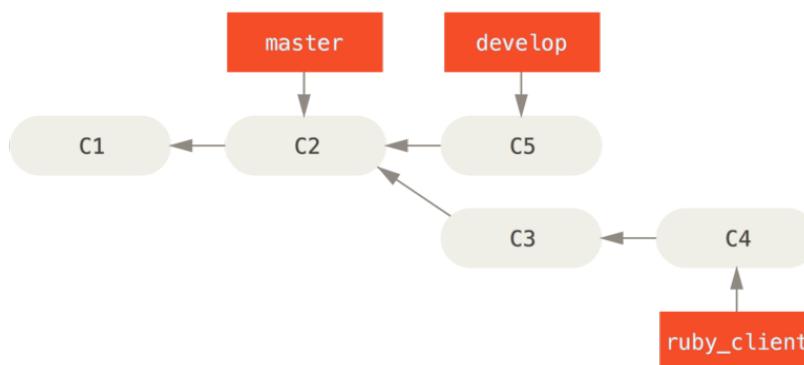
FIGURE 5-21

Tras integrar una rama puntual.

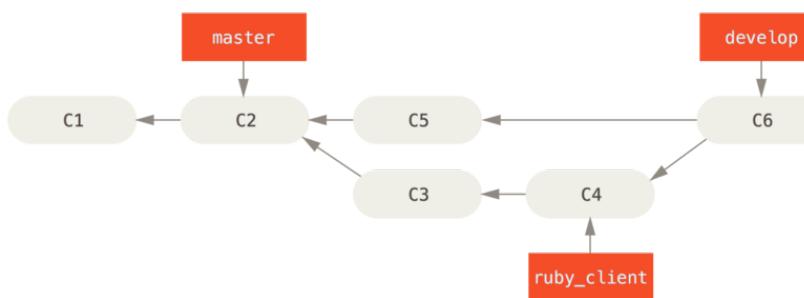


Este es probablemente el flujo de trabajo más simple y puede llegar a causar problemas si estás tratando con proyectos de mayor envergadura o más estable, donde hay que ser realmente cuidadoso al introducir cambios.

Si tienes un proyecto más importante, podrías preferir usar un ciclo de integración en dos fases. En este escenario, tienes dos ramas de largo recorrido, `master` y `develop`, y decides que la rama `master` sólo se actualiza cuando se llega a una versión muy estable y todo el código nuevo está integrado en la rama `develop`. Ambas ramas de envían habitualmente al repositorio público. Cada vez que tengas una nueva rama puntual para integrar en (Figure 5-22), primero la fusionas con la rama `develop` (Figure 5-23); luego, tras etiquetar la versión, avanzas la rama `master` hasta el punto donde se encuentre la ahora estable rama `develop` (Figure 5-24).

**FIGURE 5-22**

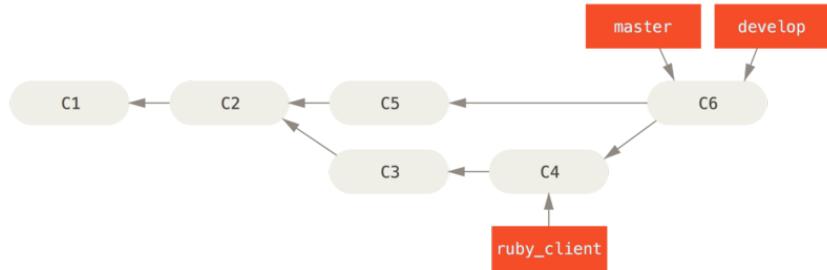
Antes de integrar una rama puntual.

**FIGURE 5-23**

Tras integrar una rama puntual.

FIGURE 5-24

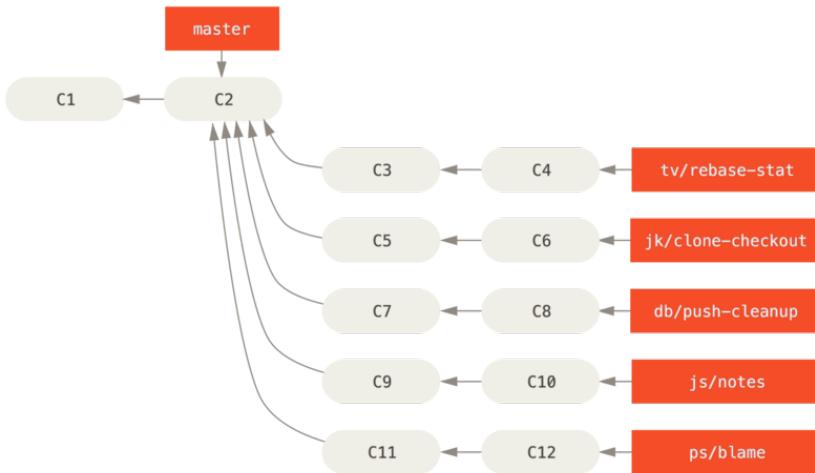
Tras el lanzamiento de una rama puntual.



De esta forma, cuando alguien clone el repositorio de tu proyecto, puede recuperar la rama `master` para construir la última versión estable y mantenerla actualizada fácilmente, o bien puede recuperar la rama `develop`, que es la que tiene los últimos desarrollos. Puedes ir un paso más allá y crear una rama de integración `integrate`, donde integres todo el trabajo. Entonces, cuando el código de esa rama sea estable y pase las pruebas, lo puedes integrar en una rama de desarrollo; y cuando se demuestre que efectivamente permanece estable durante un tiempo, avanzas la rama `master`.

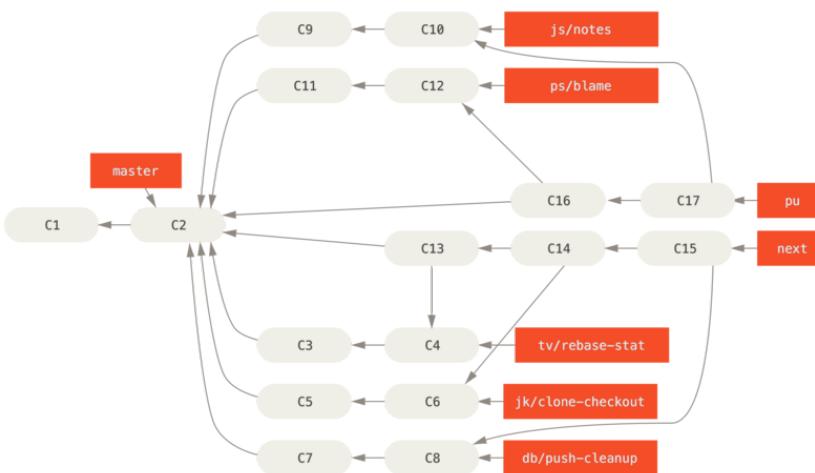
FLUJOS DE TRABAJO CON GRANDES INTEGRACIONES

El proyecto Git tiene cuatro ramas de largo recorrido: `master`, `next`, y `pu` (proposed updates, actualizaciones propuestas) para trabajos nuevos, y `maint` para trabajos de mantenimiento de versiones anteriores. Cuando los colaboradores introducen nuevos trabajos, se recopilan en ramas puntuales en el repositorio del responsable de mantenimiento, de manera similar a la que se ha descrito (ver [Figure 5-25](#)). En este punto, los nuevos trabajos se evalúan para decidir si son seguros y si están listos para los usuarios o si por el contrario necesitan más trabajo. Si son seguros, se integran en la rama `next`, y se envía dicha rama al repositorio público para que todo el mundo pueda probar las nuevas funcionalidades ya integradas.

**FIGURE 5-25**

Gestionando un conjunto complejo de ramas puntuales paralelas.

Si las nuevas funcionalidades necesitan más trabajo, se integran en la rama **pu**. Cuando se decide que estas funcionalidades son totalmente estables, se integran de nuevo en la rama **master**, construyéndolas a partir de las funcionalidades en la rama **next** que aún no habían pasado a la rama **master**. Esto significa que la rama **master** casi siempre avanza, **next** se reorganiza ocasionalmente y **pu** se reorganiza mucho más a menudo:

**FIGURE 5-26**

Fusionando ramas puntuales en ramas de integración de largo recorrido.

Cuando una rama puntual se ha integrado en la rama `master`, se elimina del repositorio. El proyecto Git también tiene una rama `maint` creada a partir de la última versión para ofrecer parches, en caso de que fuera necesaria una versión de mantenimiento. Así, cuando clonas el repositorio de Git, tienes cuatro ramas que puedes recuperar para evaluar el proyecto en diferentes etapas de desarrollo, dependiendo de si quieres tener una versión muy avanzada o de cómo quieras contribuir. De esta forma, el responsable de mantenimiento tiene un flujo de trabajo estructurado para ayudarle a aprobar las nuevas contribuciones.

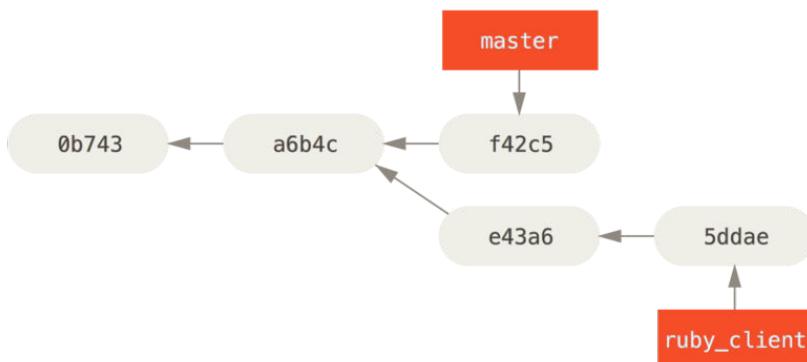
FLUJOS DE TRABAJO REORGANIZANDO O ENTRESACANDO

Otros responsables de mantenimiento prefieren reorganizar o entresacar el nuevo trabajo en su propia rama `master`, en lugar de integrarlo, para mantener un historial prácticamente lineal. Cuando tienes trabajo en una rama puntual y has decidido que quieres integrarlo, te posicionas en esa rama y ejecutas el comando `rebase` para reconstruir los cambios en tu rama `master` (o `develop`, y así sucesivamente). Si ese proceso funciona bien, puedes avanzar tu rama `master`, consiguiendo un historial lineal en tu proyecto.

Otra forma de mover trabajo de una rama a otra es entresacarlo (`cherry-pick`). En Git, “entresacar” es como hacer un `rebase` para un único commit. Toma el parche introducido en un commit e intenta reaplicarlo en la rama en la que estás actualmente. Esto es útil si tienes varios commits en una rama puntual y sólo quieres integrar uno de ellos, o si sólo tienes un commit en una rama puntual y prefieres entresacarlo en lugar de hacer una reorganización (`rebase`). Por ejemplo, imagina que tienes un proyecto como éste:

FIGURE 5-27

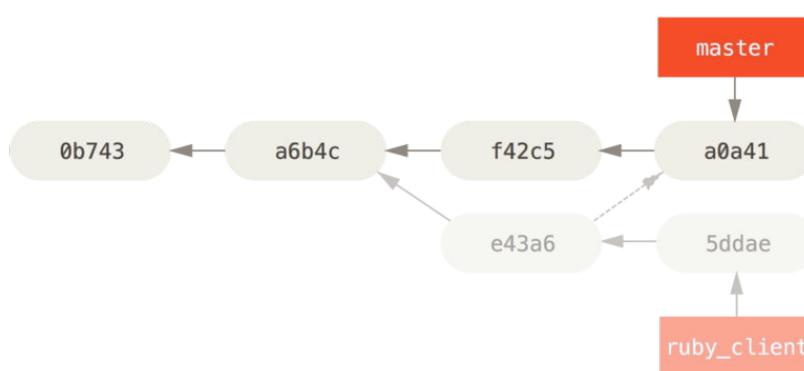
Ejemplo de historial, antes de entresacar.



Si sólo deseas integrar el commit e43a6 en tu rama `master`, puedes ejecutar

```
$ git cherry-pick e43a6fd3e94888d76779ad79fb568ed180e5fcdf
Finished one cherry-pick.
[master]: created a0a41a9: "More friendly message when locking the index fails."
  3 files changed, 17 insertions(+), 3 deletions(-)
```

Esto introduce el mismo cambio introducido en e43a6, pero genera un nuevo valor SHA-1 de confirmación, ya que la fecha en que se ha aplicado es distinta. Ahora tu historial queda así:

**FIGURE 5-28**

Historial tras entresacar un commit de una rama puntual.

En este momento ya puedes eliminar tu rama puntual y descartar los commits que no quieras integrar.

RERERE

Git dispone de una utilidad llamada “rerere” que puede resultar útil si estás haciendo muchas integraciones y reorganizaciones, o si mantienes una rama puntual de largo recorrido.

Rerere significa “reuse recorded resolution” (reutilizar resolución grabada) – es una forma de simplificar la resolución de conflictos. Cuando rerere está activo, Git mantendrá un conjunto de imágenes anteriores y posteriores a las integraciones correctas, de forma que si detecta que hay un conflicto que parece exactamente igual que otro ya corregido previamente, usará esa misma corrección sin causarte molestias.

Esta funcionalidad consta de dos partes: un parámetro de configuración y un comando. El parámetro de configuración es `rerere.enabled` y es bastante útil ponerlo en tu configuración global:

```
$ git config --global rerere.enabled true
```

Ahora, cuando hagas una integración que resuelva conflictos, la resolución se grabará en la caché por si la necesitas en un futuro.

Si fuera necesario, puedes interactuar con la cache de `rerere` usando el comando `git rerere`. Cuando se invoca sin ningún parámetro adicional, Git comprueba su base de datos de resoluciones en busca de coincidencias con cualquier conflicto durante la integración actual e intenta resolverlo (aunque se hace automáticamente en caso de que `rerere.enabled` sea `true`). También existen subcomandos para ver qué se grabará, para eliminar de la caché una resolución específica y para limpiar completamente la caché. Veremos más detalles sobre `rerere` en “[Rerere](#)”.

Etiquetando tus versiones

Cuando decides lanzar una versión, probablemente quieras etiquetarla para poder generarla más adelante en cualquier momento. Puedes crear una nueva etiqueta siguiendo los pasos descritos en [Chapter 2](#). Si decides firmar la etiqueta como responsable de mantenimiento, el etiquetado sería algo así:

```
$ git tag -s v1.5 -m 'my signed 1.5 tag'  
You need a passphrase to unlock the secret key for  
user: "Scott Chacon <schacon@gmail.com>"  
1024-bit DSA key, ID F721C45A, created 2009-02-09
```

Si firmas tus etiquetas podrías tener problemas a la hora de distribuir la clave PGP pública usada para firmarlas. El responsable de mantenimiento del proyecto Git ha conseguido solucionar este problema incluyendo su clave pública como un objeto binario en el repositorio, añadiendo a continuación una etiqueta que apunta directamente a dicho contenido. Para hacer esto, puedes averiguar qué clave necesitas lanzando el comando `gpg --list-keys`:

```
$ gpg --list-keys  
/Users/schacon/.gnupg/pubring.gpg  
-----  
pub 1024D/F721C45A 2009-02-09 [expires: 2010-02-09]
```

```
uid          Scott Chacon <schacon@gmail.com>
sub 2048g/45D02282 2009-02-09 [expires: 2010-02-09]
```

Ahora ya puedes importar directamente la clave en la base de datos de Git, exportándola y redirigiéndola a través del comando `git hash-object`, que escribe en Git un nuevo objeto binario con esos contenidos y devuelve la firma SHA-1 de dicho objeto.

```
$ gpg -a --export F721C45A | git hash-object -w --stdin
659ef797d181633c87ec71ac3f9ba29fe5775b92
```

Una vez que tienes los contenidos de tu clave en Git, puedes crear una etiqueta que apunte directamente a dicha clave indicando el nuevo valor SHA-1 que devolvió el comando `hash-object`:

```
$ git tag -a maintainer-pgp-pub 659ef797d181633c87ec71ac3f9ba29fe5775b92
```

Si ejecutas `git push --tags`, la etiqueta `maintainer-pgp-pub` será compartida con todo el mundo. Si alguien quisiera verificar una etiqueta, puede importar tu clave PGP recuperando directamente el objeto binario de la base de datos e importándolo en GPG:

```
$ git show maintainer-pgp-pub | gpg --import
```

Esa clave se puede usar para verificar todas tus etiquetas firmadas. Además, si incluyes instrucciones en el mensaje de la etiqueta, el comando `git show <tag>` permitirá que el usuario final obtenga instrucciones más específicas sobre el proceso de verificación de etiquetas.

Generando un número de compilación

Como Git no genera una serie de números monótonamente creciente como `v123` o similar con cada commit, si quieres tener un nombre más comprensible para un commit, puedes ejecutar el comando `git describe` sobre dicho commit. Git devolverá el nombre de la etiqueta más próxima junto con el número de commits sobre esa etiqueta y una parte del valor SHA-1 del commit que estás describiendo:

```
$ git describe master
v1.6.2-rc1-20-g8c5b85c
```

De esta forma, puedes exportar una instantánea o generar un nombre comprensible por cualquier persona. De hecho, si construyes Git a partir del código fuente clonado del repositorio Git, `git --version` devuelve algo parecido a esto. Si estás describiendo un commit que has etiquetado directamente, te dará el nombre de la etiqueta.

El comando `git describe` da preferencia a etiquetas anotadas (etiquetas creadas con las opciones `-a` o `-s`), por lo que las etiquetas de versión deberían crearse de esta forma si estás usando `git describe`, para garantizar que el commit es nombrado adecuadamente cuando se describe. También puedes usar esta descripción como objetivo de un comando `checkout` o `show`, aunque depende de la parte final del valor SHA-1 abreviado, por lo que podría no ser válida para siempre. Por ejemplo, recientemente el núcleo de Linux pasó de 8 a 10 caracteres para asegurar la unicidad del objeto SHA-1, por lo que los nombres antiguos devueltos por `git describe` fueron invalidados.

Preparando una versión

Ahora quieres lanzar una versión. Una cosa que querrás hacer será crear un archivo con la última instantánea del código para esas pobres almas que no usan Git. El comando para hacerlo es `git archive`:

```
$ git archive master --prefix='project/' | gzip > `git describe master`.tar.gz
$ ls *.tar.gz
v1.6.2-rc1-20-g8c5b85c.tar.gz
```

Si alguien abre el fichero tar, obtiene la última instantánea de tu proyecto bajo un directorio `project`. También puedes crear un archivo zip de la misma manera, pero añadiendo la opción `--format=zip` a `git archive`:

```
$ git archive master --prefix='project/' --format=zip > `git describe master`.zip
```

Ahora tienes tanto un archivo tar como zip con la nueva versión de tu proyecto, listos para subirlos a tu sitio web o para enviarlos por e-mail.

El registro resumido

Es el momento de enviar un mensaje a tu lista de correo informando sobre el estado de tu proyecto. Una buena opción para obtener rápidamente una especie de lista con los cambios introducidos en tu proyecto desde la última versión o e-mail es usar el comando `git shortlog`. Dicho comando resume todos los commits en el rango que se le indique; por ejemplo, el siguiente comando devuelve un resumen de todos los commits desde tu última versión, suponiendo que fuera la v1.0.1:

```
$ git shortlog --no-merges master --not v1.0.1
Chris Wanstrath (8):
    Add support for annotated tags to Grit::Tag
    Add packed-refs annotated tag support.
    Add Grit::Commit#to_patch
    Update version and History.txt
    Remove stray `puts`
    Make ls_tree ignore nils

Tom Preston-Werner (4):
    fix dates in history
    dynamic version method
    Version bump to 1.0.2
    Regenerated gemspec for version 1.0.2
```

Consigues un resumen limpio de todos los commits desde la versión v1.0.1, agrupados por autor, que puedes enviar por correo electrónico a tu lista.

Resumen

Deberías sentirte lo suficiente cómodo para contribuir en un proyecto en Git así como para mantener tu propio proyecto o integrar las contribuciones de otros usuarios. ¡Felicitaciones por ser un desarrollador eficaz con Git! En el próximo capítulo, aprenderás como usar el servicio más grande y popular para alojar proyectos de Git: Github.

GitHub 6

GitHub es el mayor proveedor de alojamiento de repositorios Git, y es el punto de encuentro para que millones de desarrolladores colaboren en el desarrollo de sus proyectos. Un gran porcentaje de los repositorios Git se almacenan en GitHub, y muchos proyectos de código abierto lo utilizan para hospedar su Git, realizar su seguimiento de fallos, hacer revisiones de código y otras cosas. Por tanto, aunque no sea parte directa del proyecto de código abierto de Git, es muy probable que durante tu uso profesional de Git necesites interactuar con GitHub en algún momento.

Este capítulo trata del uso eficaz de GitHub. Veremos cómo crear y gestionar una cuenta, crear y gestionar repositorios Git, también los flujos de trabajo (workflows) habituales para participar en proyectos y para aceptar nuevos participantes en los tuyos, la interfaz de programación de GitHub (API) y muchos otros pequeños trucos que te harán, en general, la vida más fácil.

Si no vas a utilizar GitHub para hospedar tus proyectos o para colaborar con otros, puedes saltar directamente a [Chapter 7](#).

CAMBIOS EN LA INTERFAZ

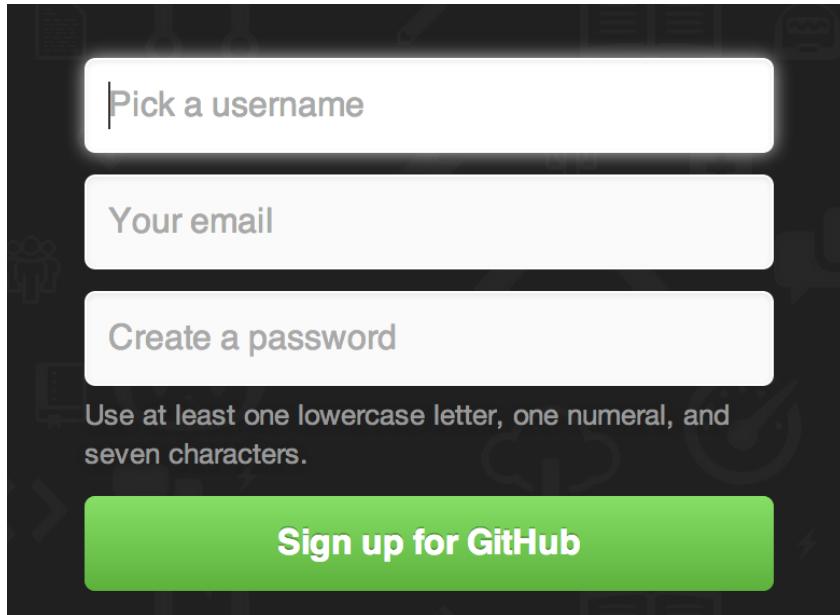
Observa que como muchos sitios web activos, el aspecto de la interfaz de usuario puede cambiar con el tiempo, frente a las capturas de pantalla que incluye este libro. Probablemente la versión en línea de este libro tenga esas capturas más actualizadas.

Creación y configuración de la cuenta

Lo primero que necesitas es una cuenta de usuario gratuita. Simplemente visita <https://github.com>, elige un nombre de usuario que no esté ya en uso, proporciona un correo y una contraseña, y pulsa el botón verde grande “Sign up for GitHub”.

FIGURE 6-1

Formulario para darse de alta en GitHub.



Lo siguiente que verás es la página de precios para planes mejores, pero lo puedes ignorar por el momento. GitHub te enviará un correo para verificar la dirección que les has dado. Confirma la dirección ahora, es bastante importante (como veremos después).

GitHub proporciona toda su funcionalidad en cuentas gratuitas, con la limitación de que todos tus proyectos serán públicos (todos los usuarios tendrán acceso de lectura). Los planes de pago de GitHub te permite tener algunos proyectos privados, pero esto es algo que no veremos en este libro.

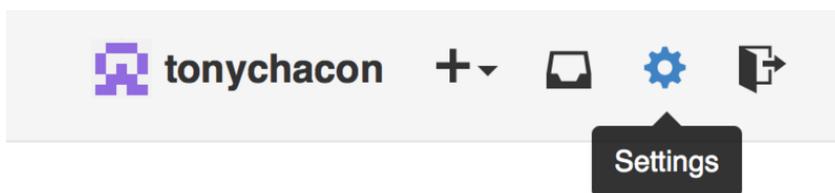
Si pulsas en el logo del gato con patas de pulpo en la parte superior izquierda de la pantalla llegarás a tu escritorio principal. Ahora ya estás listo para comenzar a usar GitHub.

Acceso SSH

Desde ya, puedes acceder a los repositorios Git utilizando el protocolo <https://>, identificándote con el usuario y la contraseña que acabas de elegir. Sin embargo, para simplificar el clonado de proyectos públicos, no necesitas

crearte la cuenta. Es decir, la cuenta solo la necesitas cuando comienzas a hacer cosas como bifurcar (fork) proyectos y enviar tus propios cambios más tarde.

Si prefieres usar SSH, necesitas configurar una clave pública. Si aun no la tienes, mira cómo generarla en “**Generando tu clave pública SSH**.”) Abre tu panel de control de la cuenta utilizando el enlace de la parte superior derecha de la ventana:

**FIGURE 6-2**

Enlace “Account settings”.

Aquí selecciona en el lado izquierdo la opción “SSH keys”.

A screenshot of the "SSH Keys" section of the GitHub account settings. On the left, a sidebar menu has "SSH keys" selected. The main area shows a message: "Need help? Check out our guide to generating SSH keys or troubleshoot common SSH Problems". Below this is a table titled "SSH Keys" with one row: "There are no SSH keys with access to your account." To the right of the table is a "Add SSH key" button. Below the table is a form titled "Add an SSH Key" with fields for "Title" (a text input field) and "Key" (a large text area). At the bottom of the form is a green "Add key" button.

FIGURE 6-3

Enlace “SSH keys”.

Desde ahí, pulsa sobre “Add an SSH key”, proporcionando un nombre y pegando los contenidos del fichero `~/.ssh/id_rsa.pub` (o donde hayas definido tu clave pública) en el área de texto, y pulsa sobre “Add key”.

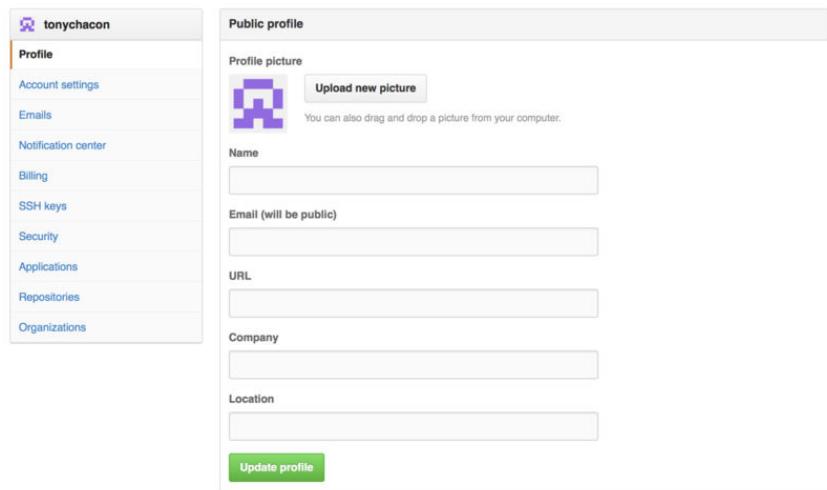
Asegúrate de darle a tu clave un nombre que puedas recordar. Puedes, por ejemplo, añadir claves diferentes, con nombres como “Clave Portátil” o “Cuenta de trabajo” de modo que si tienes que revocar alguna clave más tarde, te resultará más fácil decidir cuál es.

Tu ícono

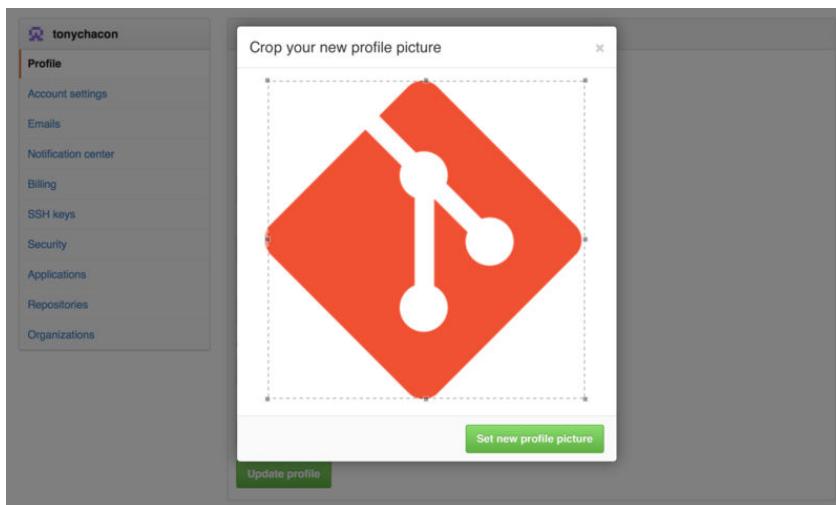
También, si quieres, puedes reemplazar el ícono (avatar) que te generaron para ti con una imagen de tu elección. En primer lugar selecciona la opción “Profile” (encima de la opción de “SSH keys”) y pulsa sobre “Upload new picture”.

FIGURE 6-4

Enlace “Profile”.



Nosotros elegiremos como ejemplo una copia del logo de Git que tengamos en el disco duro y luego tendremos opción de recortarlo al subirlo.

**FIGURE 6-5***Recortar tu ícono*

Desde ahora, quien vea tu perfil o tus contribuciones a repositorios verá tu nuevo ícono junto a tu nombre.

Si da la casualidad que ya tienes tu ícono en el popular servicio Gravatar (conocido por su uso en las cuentas de Wordpress), este ícono será detectado y no tendrás que hacer, si quieres, este paso.

Tus direcciones de correo

La forma con la que GitHub identifica tus contribuciones a Git es mediante la dirección de correo electrónico. Si tienes varias direcciones diferentes en tus contribuciones (commits) y quieres que GitHub sepa que son de tu cuenta, necesitas añadirlas todas en la sección Emails de la sección de administración.

FIGURE 6-6

Añadiendo direcciones de correo

The screenshot shows the 'Emails' section of the GitHub account settings for user 'tonychacon'. The primary email address 'tonychacon@example.com' is listed with the status 'Primary' and 'Public'. Below it are two other email addresses: 'tchacon@example.com' and 'tony.chacon@example.com', both of which are marked as 'Unverified'. There is a button to 'Send verification email' next to the unverified addresses. A link to 'Set as primary' is also present for the unverified addresses. At the bottom, there is a field to 'Add email address' and a checkbox to 'Keep my email address private', with a note explaining that GitHub will use the user's GitHub email for operations.

En **Figure 6-6** podemos ver los diferentes estados posibles. La dirección inicial se verifica y se utiliza como dirección principal, lo que significa que es donde vas a recibir cualquier notificación. La siguiente dirección se puede verificar y ponerla entonces como dirección principal, si quieres cambiarla. La última dirección no está verificada, lo que significa que no puedes usarla como principal. Pero si GitHub ve un commit con esa dirección, la identificará asociándola a tu usuario.

Autentificación de dos pasos

Finalmente, y para mayor seguridad, deberías configurar la Autentificación de Dos Pasos o “2FA”. Este tipo de autentificación se está haciendo más popular para reducir el riesgo de que te roben la cuenta. Al activarla, GitHub te pedirá identificarte de dos formas, de forma que si una de ellas resulta comprometida, el atacante no conseguirá acceso a tu cuenta.

Puedes encontrar la configuración de “2FA” en la opción Security de los ajustes de la cuenta.

FIGURE 6-7

2FA dentro de Security

Si pulsas en el botón “Set up two-factor authentication”, te saldrá una página de configuración donde podrás elegir un generador de códigos en una aplicación de móvil (es decir, códigos de un solo uso) o bien podrás elegir que te envíen un SMS cada vez que necesites entrar.

Cuando configures este método de autenticación, tu cuenta será un poco más segura ya que tendrás que proporcionar un código junto a tu contraseña cada vez que accedas a GitHub.

Participando en Proyectos

Una vez que tienes la cuenta configurada, veremos algunos detalles útiles para ayudarte a participar en proyectos existentes.

Bifurcación (fork) de proyectos

Si quieras participar en un proyecto existente, en el que no tengas permisos de escritura, puedes bifurcarlo (hacer un “fork”). Esto consiste en crear una copia completa del repositorio totalmente bajo tu control: se encontrará en tu cuenta y podrás escribir en él sin limitaciones.

Históricamente, el término “fork” podía tener connotaciones algo negativas, ya que significaba que alguien realizaba una copia del código fuente del proyecto y las comenzaba a modificar de forma independiente al proyecto original, tal vez para crear un proyecto competidor y dividir a su comunidad de colaboradores. En GitHub, el “fork” es simplemente una copia del repositorio donde puedes escribir, haciendo públicos tus propios cambios, como una manera abierta de participación.

De esta forma, los proyectos no necesitan añadir colaboradores con acceso de escritura (push). La gente puede bifurcar un proyecto, enviar sus propios cambios a su copia y luego remitir esos cambios al repositorio original para su aprobación, creando lo que se llama un Pull Request, que veremos más adelante. Esto permite abrir una discusión para la revisión del código, donde propietario y participante pueden comunicarse acerca de los cambios y, en última instancia, el propietario original puede aceptarlos e integrarlos en el proyecto original cuando lo considere adecuado.

Para bifurcar un proyecto, visita la página del mismo y pulsa sobre el botón “Fork” del lado superior derecho de la página.

FIGURE 6-8

Botón “Fork”.



En unos segundos te redireccionarán a una página nueva de proyecto, en tu cuenta y con tu propia copia del código fuente.

El Flujo de Trabajo en GitHub

GitHub está diseñado alrededor de un flujo de trabajo de colaboración específico, centrado en las solicitudes de integración (“pull request”). Este flujo es válido tanto si colaboras con un pequeño equipo en un repositorio compartido como si lo haces en una gran red de participantes con docenas de bifurcaciones particulares. Se centra en el workflow “**Ramas Puntuales**” cubierto en **Chapter 3**.

El funcionamiento habitual es el siguiente:

1. Se crea una rama a partir de `master`.
2. Se realizan algunos commits hacia esa rama.
3. Se envía esa rama hacia tu copia (fork) del proyecto.

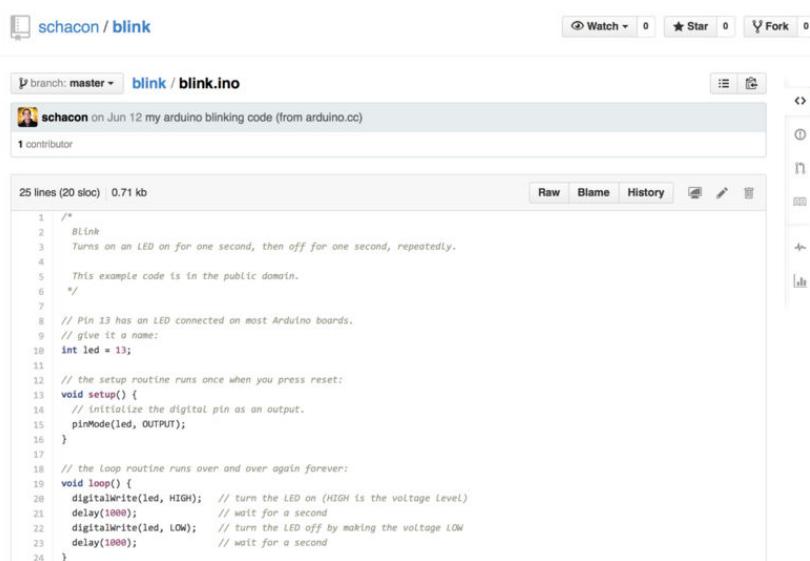
4. Abres un Pull Request en GitHub.
5. Se participa en la discusión asociada y, opcionalmente, se realizan nuevos commits.
6. El propietario del proyecto original cierra el Pull Request, bien fusionando la rama con tus cambios o bien rechazándolos.

Este es, básicamente, el flujo de trabajo del Responsable de Integración visto en “**Integration-Manager Workflow**”, pero en lugar de usar el correo para comunicarnos y revisar los cambios, lo que se hace es usar las herramientas web de GitHub.

Veamos un ejemplo de cómo proponer un cambio en un proyecto de código abierto hospedado en GitHub, utilizando esta forma de trabajar.

CREACIÓN DEL PULL REQUEST

Tony está buscando código para ejecutar en su microcontrolador Arduino, y ha encontrado un programa interesante en GitHub, en la dirección <https://github.com/schacon/blink>.



The screenshot shows the GitHub repository page for the user schacon named 'blink'. The repository has 0 stars and 0 forks. The master branch is selected, showing the file 'blink/blink.ino'. The code is as follows:

```

25 lines (20 sloc) 0.71 kb
/*
  Blink
  Turns on an LED on for one second, then off for one second, repeatedly.

  This example code is in the public domain.

  // Pin 13 has an LED connected on most Arduino boards.
  // give it a name:
  int led = 13;

  // the setup routine runs once when you press reset:
void setup() {
  // initialize the digital pin as an output:
  pinMode(led, OUTPUT);
}

  // the Loop routine runs over and over again forever:
void loop() {
  digitalWrite(led, HIGH); // turn the LED on (HIGH is the voltage Level)
  delay(1000);           // wait for a second
  digitalWrite(led, LOW); // turn the LED off by making the voltage LOW
  delay(1000);           // wait for a second
}

```

FIGURE 6-9

El proyecto en el que queremos participar.

El único problema es que la velocidad de parpadeo es muy rápida, y piensa que es mucho mejor esperar 3 segundos en lugar de 1 entre cada cambio de

estado. Luego, nuestra mejora consistirá en cambiar la velocidad y enviar el cambio al proyecto como un cambio propuesto.

Lo primero que se hace, es pulsar en el botón *Fork* ya conocido para hacer nuestra propia copia del proyecto. Nuestro nombre de usuario es “tonychacon” por lo que la copia del proyecto tendrá como dirección <https://github.com/tonychacon/blink>, y en esta copia es donde podemos trabajar. La clonaremos localmente, crearemos una rama, realizaremos el cambio sobre el código fuente y finalmente enviaremos esos cambios a GitHub.

```
$ git clone https://github.com/tonychacon/blink ❶
Cloning into 'blink'...

$ cd blink
$ git checkout -b slow-blink ❷
Switched to a new branch 'slow-blink'

$ sed -i '' 's/1000/3000/' blink.ino ❸

$ git diff --word-diff ❹
diff --git a/blink.ino b/blink.ino
index 15b9911..a6cc5a5 100644
--- a/blink.ino
+++ b/blink.ino
@@ -18,7 +18,7 @@ void setup() {
// the loop routine runs over and over again forever:
void loop() {
    digitalWrite(led, HIGH);      // turn the LED on (HIGH is the voltage level)
[-delay(1000);-]{+delay(3000);+}          // wait for a second
    digitalWrite(led, LOW);       // turn the LED off by making the voltage LOW
[-delay(1000);-]{+delay(3000);+}          // wait for a second
}

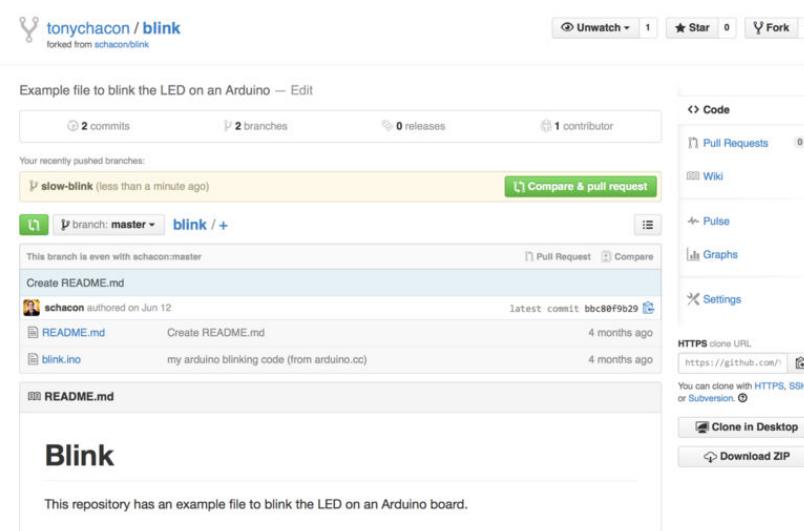
$ git commit -a -m 'three seconds is better' ❺
[slow-blink 5ca509d] three seconds is better
 1 file changed, 2 insertions(+), 2 deletions(-)

$ git push origin slow-blink ❻
Username for 'https://github.com': tonychacon
Password for 'https://tonychacon@github.com':
Counting objects: 5, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 340 bytes | 0 bytes/s, done.
Total 3 (delta 1), reused 0 (delta 0)
To https://github.com/tonychacon/blink
 * [new branch]      slow-blink -> slow-blink
```

- ① Clonar nuestro fork en nuestro equipo
- ② Crear la rama, que sea descriptiva
- ③ Realizar nuestros cambios
- ④ Comprobar los cambios
- ⑤ Realizar un commit de los cambios en la rama
- ⑥ Enviar nuestra nueva rama de vuelta a nuestro fork

Ahora, si miramos nuestra bifurcación en GitHub, veremos que aparece un aviso de creación de la rama y nos dará la oportunidad de hacer una solicitud de integración con el proyecto original.

También puedes ir a la página “Branches” en <https://github.com/<user>/<project>/branches> para localizar la rama y abrir el Pull Request desde ahí.

**FIGURE 6-10**

Botón Pull Request

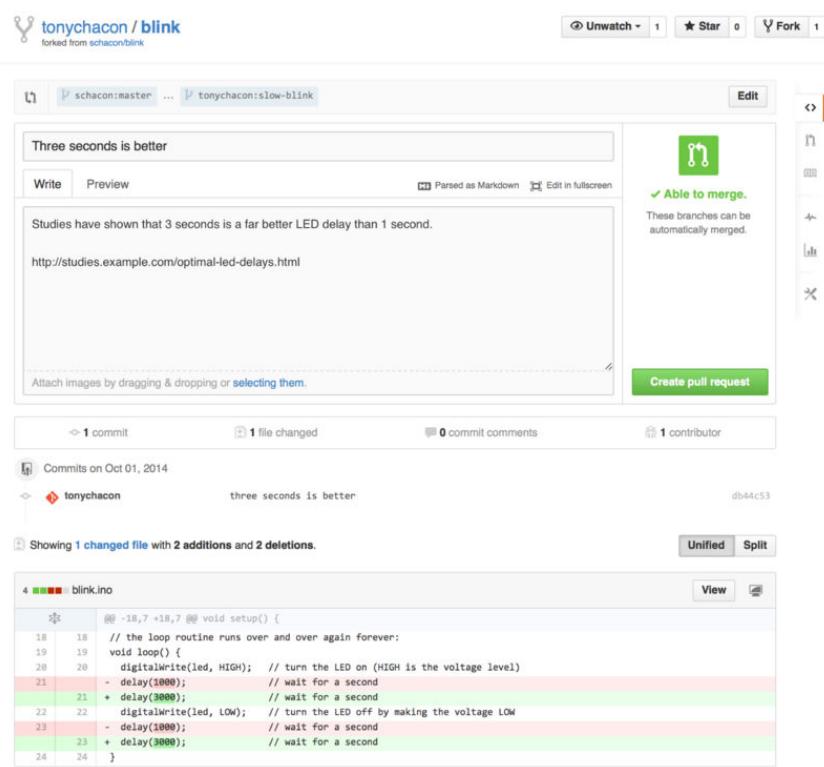
Si pulsamos en el botón verde, veremos una pantalla que permite crear un título y una descripción para darle al propietario original una buena razón para tenerla en cuenta. Normalmente debemos realizar cierto esfuerzo en hacer una

buenas descripciones para que el autor sepa realmente qué estamos aportando y lo valore adecuadamente.

También veremos la lista de commits de la rama que están “por delante” de la rama `master` (en este caso, la única) y un diff unificado de los cambios que se aplicarían si se fusionasen con el proyecto original.

FIGURE 6-11

Página de creación del Pull Request



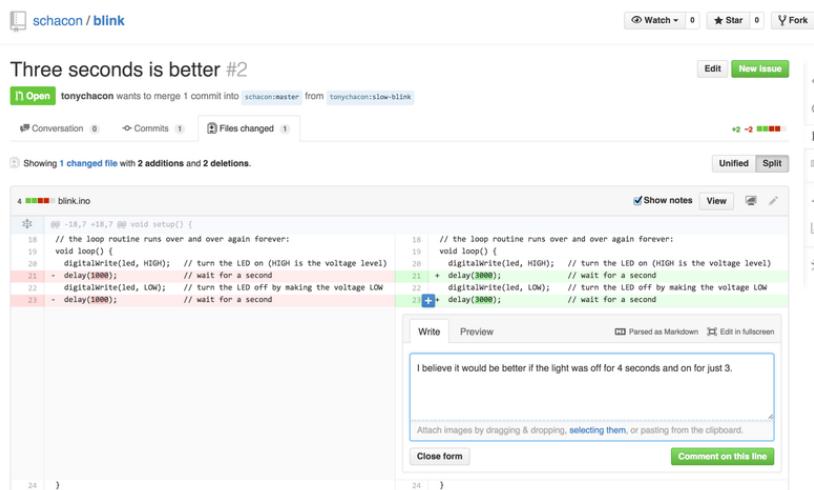
Cuando seleccionas el botón *Create pull request*, el propietario del proyecto que has bifurcado recibirá una notificación de que alguien sugiere un cambio junto a un enlace donde está toda la información.

Aunque los Pull Request se utilizan en proyectos públicos como este donde el ayudante tiene un conjunto de cambios completos para enviar, también se utiliza en proyectos internos al principio del ciclo de desarrollo: puedes crear el Pull Request con una rama propia y seguir enviando commits a dicha rama después de crear el Pull Request, siguiendo un modelo iterativo de desarrollo, en lugar de crear la rama cuando ya has finalizado todo el trabajo.

EVOLUCIÓN DEL PULL REQUEST

En este punto, el propietario puede revisar el cambio sugerido e incorporarlo (merge) al proyecto, o bien rechazarlo o comentarlo. Por ejemplo, si le gusta la idea pero prefiere esperar un poco.

La discusión, en los workflow de **Chapter 5**, tiene lugar por correo electrónico, mientras que en GitHub tiene lugar en línea. El propietario del proyecto puede revisar el diff y dejar un comentario pulsando en cualquier línea del diff.



The screenshot shows a GitHub pull request for a file named 'blink.ino'. The commit message is 'Three seconds is better #2'. The code diff shows changes made by 'tonychacon' to the 'loop()' function. A specific line of code, line 21, is highlighted with a green background, indicating it is being commented on. A comment box is open over this line with the text: 'I believe it would be better if the light was off for 4 seconds and on for just 3.' The GitHub interface includes standard navigation and commit history buttons.

FIGURE 6-12

Comentando una
línea concreta del
diff

Cuando el responsable hace el comentario, la persona que solicitó la integración (y otras personas que hayan configurado sus cuentas para escuchar los cambios del repositorio) recibirán una notificación. Más tarde veremos cómo personalizar esto, pero si las notificaciones están activas, Tony recibiría un correo como este:

FIGURE 6-13

Comentarios enviados en notificaciones de correo

Re: [blink] Three seconds is better (#2) 

Scott Chacon <notifications@github.com>
to schacon/blink, me 

10:55 AM (18 minutes ago)   

In blink.ino:

```
>   digitalWrite(led, LOW);      // turn the LED off by making the voltage LOW
> -   delay(1000);              // wait for a second
> +   delay(3000);              // wait for a second
```

I believe it would be better if the light was off for 4 seconds and on for just 3.

—
Reply to this email directly or [view it on GitHub](#).

FIGURE 6-14

Página de discusión del Pull Request

Three seconds is better #2  

 tonychacon wants to merge 1 commit into schacon:master from tonychacon:slow-blink

Conversation 1 Commits 1 Files changed 1

 tonychacon commented 6 minutes ago
Studies have shown that 3 seconds is a far better LED delay than 1 second.
<http://studies.example.com/optimal-led-delays.html>

 three seconds is better db44c53
I believe it would be better if the light was off for 4 seconds and on for just 3.

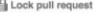
 schacon commented on the diff just now
View full changes
blink.ino
((6 lines not shown))
22 22 digitalWrite(led, LOW); // turn the LED off by making the voltage LOW
23 - delay(1000); // wait for a second
23 + delay(3000); // wait for a second

 schacon added a note just now
I believe it would be better if the light was off for 4 seconds and on for just 3.

Add a line note

 schacon commented just now
If you make that change, I'll be happy to merge this.

Labels None yet
Milestone No milestone
Assignee No one—assign yourself

Notifications 
You're receiving notifications because you commented.
2 participants
 Lock pull request

El participante puede ver ahora qué tiene que hacer para ver aceptado su cambio. Con suerte sera poco trabajo. Mientras que con el correo electrónico

tendrías que revisar los cambios y reenviarlos a la lista de correo, en GitHub puedes, simplemente, enviar un nuevo commit a la rama y subirla (push).

Si el participante hace esto, el coordinador del proyecto será notificado nuevamente y, cuando visiten la página, verán lo que ha cambiado. De hecho, al ver que un cambio en una línea de código tenía ya un comentario, GitHub se da cuenta y oculta el diff obsoleto.

Three seconds is better #2

The screenshot shows a GitHub pull request page. At the top, it says "tonychacon wants to merge 3 commits into schacon:master from tonychacon:slow-blink". Below this, there are tabs for "Conversation" (3), "Commits" (3), and "Files changed" (1). The "Files changed" tab is selected, showing a single file named "three seconds is better". A comment from "tonychacon" states: "Studies have shown that 3 seconds is a far better LED delay than 1 second." with a link to "http://studies.example.com/optimal-led-delays.html". A comment from "schacon" follows, saying: "If you make that change, I'll be happy to merge this." Below the file changes, "tonychacon" has added three commits: "longer off time" (commit db44c53) and "remove trailing whitespace" (commit ef4725c). The latest comment from "tonychacon" says: "I changed it to 4 seconds and also removed some trailing whitespace that I found. Anything else you would like me to do?" At the bottom of the "Files changed" section, a green box states: "This pull request can be automatically merged. You can also merge branches on the command line." with a "Merge pull request" button.

FIGURE 6-15

Pull Request final

Es interesante notar que si pulsas en “Files Changed” dentro del Pull Request, verás el “diff unificado”, es decir, los cambios que se introducirían en la rama principal si la otra rama fuera fusionada. En términos de git, lo que hace es mostrarte la salida del comando `git diff master ... <rama>`. Mira en “Decidiendo qué introducir” para saber más sobre este tipo de diff.

Otra cosa interesante es que GitHub también comprueba si el Pull Request se fusionaría limpiamente (de forma automática) dando entonces un botón para hacerlo. Este botón solo lo veremos si además somos los propietarios del repositorio. Si pulsas este botón, GitHub fusionará sin avance rápido, es decir, que incluso si la fusión pudiera ser de tipo avance-rápido, de todas formas crearía un commit de fusión.

Si quieres, puedes obtener la rama en tu equipo y hacer la fusión localmente. Si fusionas esta rama en la rama `master` y la subes a GitHub, el Pull Request se cerrará de forma automática.

Este es el flujo de trabajo básico que casi todos los proyectos de GitHub utilizan. Se crean las ramas de trabajo, se crean con ellas los Pull Requests, se genera una discusión, se añade probablemente más trabajo a la rama y finalmente la petición es cerrada (rechazada) o fusionada.

NO SOLO FORKS

Observa que también puedes abrir un Pull Request entre dos ramas del mismo repositorio. Si estás trabajando en una característica con alguien y ambos tenéis acceso de escritura al repositorio, puedes subir una rama al mismo y abrir un Pull Request con ella de fusión con `master` para poder formalizar el proceso de revisión de código y discusión. Para esto no se requieren bifurcaciones (forks).

Pull Requests Avanzados

Ahora que sabemos cómo participar de forma básica en un proyecto de GitHub, veamos algunos trucos más acerca de los Pull Requests que ayudarán a usarlos de forma más eficaz.

PULL REQUESTS COMO PARCHES

Hay que entender que muchos proyectos no tienen la idea de que los Pull Requests sean colas de parches perfectos que se pueden aplicar limpiamente en orden, como sucede con los proyectos basados en listas de correo. Casi todos los proyectos de GitHub consideran las ramas de Pull Requests como conversaciones evolutivas acerca de un cambio propuesto, culminando en un diff unificado que se aplica fusionando.

Esto es importante, ya que normalmente el cambio se sugiere bastante antes de que el código sea suficientemente bueno, lo que lo aleja bastante del modelo basado en parches por lista de correo. Esto facilita una discusión más temprana con los colaboradores, lo que hace que la llegada de la solución correcta sea un esfuerzo de comunidad. Cuando el cambio llega con un Pull Request y los colaboradores o la comunidad sugieren un cambio, normalmente

los parches no son directamente alterados, sino que se realiza un nuevo commit en la rama para enviar la diferencia que materializa esas sugerencias, haciendo avanzar la conversación con el contexto del trabajo previo intacto.

Por ejemplo, si miras de nuevo en **Figure 6-15**, verás que el colaborador no reorganiza su commit y envía un nuevo Pull Request. En lugar, lo que hace es añadir nuevos commits y los envía a la misma rama. De este modo, si vuelves a mirar el Pull Request en el futuro, puedes encontrar fácilmente todo el contexto con todas las decisiones tomadas. Al pulsar el botón “Merge”, se crea un commit de fusión que referencia al Pull Request, con lo que es fácil localizar para revisar la conversación original, si es necesario.

MANTENIÉNDONOS ACTUALIZADOS

Si el Pull Request se queda anticuado o por cualquier otra razón no puede fusionarse limpiamente, lo normal es corregirlo para que el responsable pueda fusionarlo fácilmente. GitHub comprobará esto y te dirá si cada Pull Request tiene una fusión trivial posible o no.



FIGURE 6-16

Pull Request que no puede fusionarse limpiamente

Si ves algo parecido a **Figure 6-16**, seguramente prefieras corregir la rama de forma que se vuelva verde de nuevo y el responsable no tenga trabajo extra con ella.

Tienes dos opciones para hacer esto. Puedes, por un lado, reorganizar (rebase) la rama con el contenido de la rama master (normalmente esta es la rama desde donde se hizo la bifurcación), o bien puedes fusionar la rama objetivo en la tuya.

Muchos desarrolladores eligen la segunda opción, por las mismas razones que que dijimos en la sección anterior. Lo que importa aquí es la historia y la fusión final, por lo que reorganizar no es mucho más que tener una historia más limpia y, sin embargo, es de lejos más complicado de hacer y con más posibilidad de error.

Si quieras fusionar en la rama objetivo para hacer que tu Pull Request sea fusionable, deberías añadir el repositorio original como un nuevo remoto, bájartelo (fetch), fusionar la rama principal en la tuya, corregir los problemas que surjan y finalmente enviarla (push) a la misma rama donde hiciste la solicitud de integración.

Por ejemplo, supongamos que en el ejemplo “tonychacon” que hemos venido usando, el autor original hace un cambio que crea un conflicto con el Pull Request. Seguiremos entonces los siguientes pasos.

```
$ git remote add upstream https://github.com/schacon/blink ①
$ git fetch upstream ②
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (3/3), done.
Unpacking objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
From https://github.com/schacon/blink
 * [new branch]      master      -> upstream/master

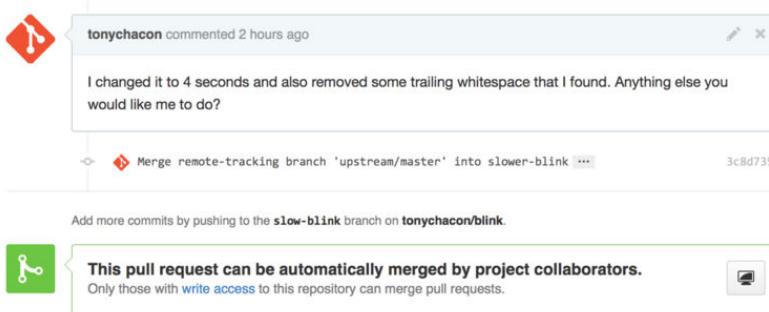
$ git merge upstream/master ③
Auto-merging blink.ino
CONFLICT (content): Merge conflict in blink.ino
Automatic merge failed; fix conflicts and then commit the result.

$ vim blink.ino ④
$ git add blink.ino
$ git commit
[slow-blink 3c8d735] Merge remote-tracking branch 'upstream/master' \
    into slower-blink

$ git push origin slow-blink ⑤
Counting objects: 6, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 682 bytes | 0 bytes/s, done.
Total 6 (delta 2), reused 0 (delta 0)
To https://github.com/tonychacon/blink
    ef4725c..3c8d735  slower-blink -> slow-blink
```

- ① Añadir el repositorio original como un remoto llamado “upstream”
- ② Obtener del remoto lo último enviado al repositorio
- ③ Fusionar la rama principal en la nuestra
- ④ Corregir el conflicto surgido
- ⑤ Enviar de nuevo los cambios a la rama del Pull Request

Cuando haces esto, el Pull Request se actualiza automáticamente y se rechequea para ver si es posible un fusionado automático o no.

**FIGURE 6-17**

Ahora el Pull Request ya fusiona bien

Una de las cosas interesantes de Git es que puedes hacer esto continuamente. Si tienes un proyecto con mucha historia, puedes fácilmente fusionarte la rama objetivo (`master`) una y otra vez cada vez que sea necesario, evitando conflictos y haciendo que el proceso de integración de tus cambios sea muy manejable.

Si finalmente prefieres reorganizar la rama para limpiarla, también puedes hacerlo, pero se recomienda no forzar el push sobre la rama del Pull Request. Si otras personas se la han bajado y hacen más trabajo en ella, provocarás los problemas vistos en “**Los Peligros de Reorganizar**”. En su lugar, envía la rama reorganizada a una nueva rama de GitHub y abre con ella un nuevo Pull Request, con referencia al antiguo, cerrando además éste.

REFERENCIAS

La siguiente pregunta puede ser “¿cómo hago una referencia a un Pull Request antiguo?”. La respuesta es, de varias formas.

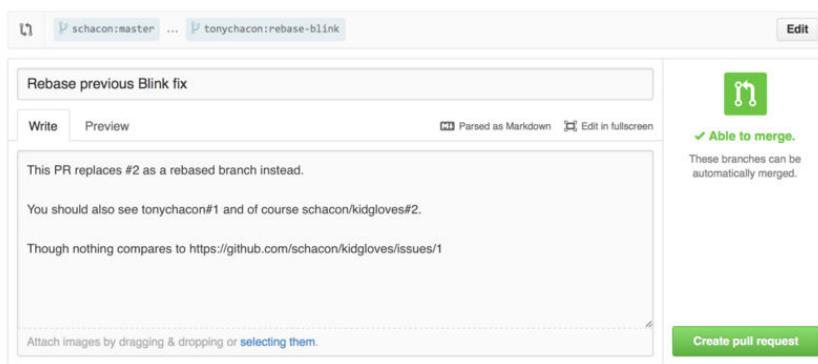
Comencemos con cómo referenciar otro Pull Request o una incidencia (Issue). Todas las incidencias y Pull Requests tienen un número único que los identifica. Este número no se repite dentro de un mismo proyecto. Por ejemplo, dentro de un proyecto solo podemos tener un Pull Request con el número 3, y una incidencia con el número 3. Si quieres hacer referencia al mismo, basta con poner el símbolo # delante del número, en cualquier comentario o descripción del Pull Request o incidencia. También se puede poner referencia tipo `usuario#numero` para referirnos a un Pull Request o incidencia en una bifurcación que haya creado ese usuario, o incluso puede usarse la forma `usuario/repo#num` para referirse a una incidencia o Pull Request en otro repositorio diferente.

Veamos un ejemplo. Supongamos que hemos reorganizado la rama del ejemplo anterior, creado un nuevo pull request para ella y ahora queremos hac-

er una referencia al viejo Pull Request desde el nuevo. También queremos hacer referencia a una incidencia en la bifurcación del repositorio, y una incidencia de un proyecto totalmente distinto. Podemos llenar la descripción justo como vemos en **Figure 6-18**.

FIGURE 6-18

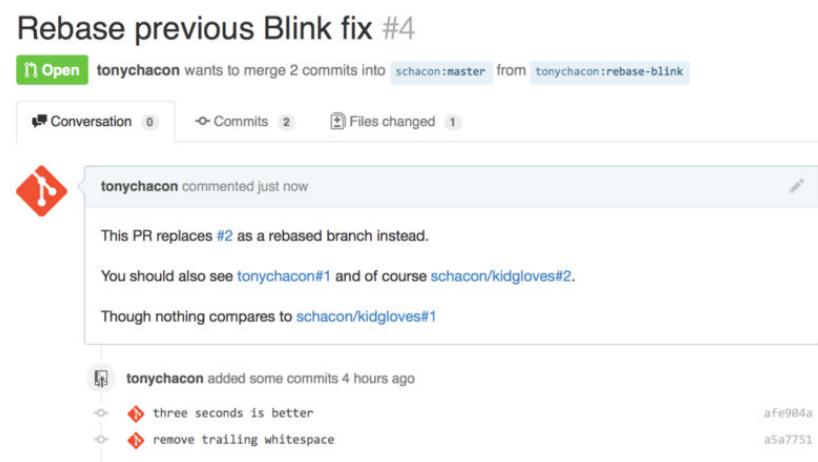
Referencias cruzadas en un Pull Request.



Cuando enviamos este pull request, veremos todo como en **Figure 6-19**.

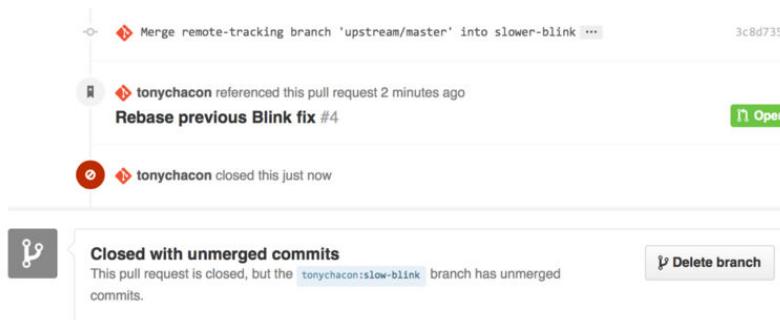
FIGURE 6-19

Cómo se ven las referencias cruzadas en el Pull Request.



Observa que la URL completa de GitHub que hemos puesto ahí ha sido acortada a la información que necesitamos realmente.

Ahora, si Tony regresa y cierra el Pull Request original, veremos que GitHub crea un evento en la línea de tiempo del Pull Request. Esto significa que cualquier que visite este Pull Request y vea que está cerrado, puede fácilmente enlazarlo al que lo hizo obsoleto. El enlace se mostrará tal como en **Figure 6-20**.

**FIGURE 6-20**

Cómo se ven las referencias cruzadas en el Pull Request.

Además de los números de incidencia, también puedes hacer referencia a un commit específico usando la firma SHA-1. Puedes utilizar la cadena SHA-1 completa (de 40 caracteres) y al detectarla GitHub en un comentario, la convertirá automáticamente en un enlace directo al commit. Nuevamente, puedes hacer referencia a commits en bifurcaciones o en otros repositorios del mismo modo que hicimos con las incidencias.

Markdown

En enlazado a otras incidencias es solo el comienzo de las cosas interesantes que se pueden hacer con cualquier cuadro de texto de GitHub. En las descripciones de las incidencias y los Pull Requests, así como en los comentarios, y otros cuadros de texto, se puede usar lo que se conoce “formato Markdown de GitHub”. El formato Markdown es como escribir en texto plano pero que luego se convierte en texto con formato.

Mira en **Figure 6-21** un ejemplo de cómo los comentarios o el texto puede escribirse y luego formatearse con Markdown.

FIGURE 6-21

Ejemplo de texto en Markdown y cómo queda después.

The screenshot shows two parts of the GitHub interface. On the left is a 'Markdown Example' editor with the following content:

```
There is a "big" problem with the blink code. Not with the idea, but with the _code_.

## What is the problem?

As you can see [here](https://github.com/tonychacon/blink/blob/master/blink.ino#L10), the LED uses the number 13 which has the following issues:

* It is unlucky
* It is two decimal places

The if we replace `int led = 13;` with `int led = 7;` it will be far more lucky.

As Kanye West said:

>We're living the future so
>the present is our past.

git logo([https://logos.example.com/git-logo.png])
```

On the right is a comment from user tonychacon:

tonychacon commented just now
There is a **big** problem with the blink code. Not with the idea, but with the code.
What is the problem?
As you can see [here](#), the LED uses the number 13 which has the following issues:
• It is unlucky
• It is two decimal places
The if we replace `int led = 13;` with `int led = 7;`, it will be far more lucky.
As Kanye West said:
We're living the future so
the present is our past.

EL FORMATO MARKDOWN DE GITHUB

En GitHub se añaden algunas cosas a la sintaxis básica del Markdown. Son útiles al tener relación con los Pull Requests o las incidencias.

Listas de tareas

La primera característica añadida, especialmente interesante para los Pull Requests, son las listas de tareas. Una lista de tareas es una lista de cosas con su marcador para indicar que han terminado. En un Pull Requests o una incidencia nos sirven para anotar la lista de cosas pendientes para considerar terminado el trabajo relacionado con esa incidencia.

Puedes crear una lista de tareas así:

- [X] Write the code
- [] Write all the tests
- [] Document the code

Si incluimos esto en la descripción de nuestra incidencia o Pull Request, lo veremos con el aspecto de **Figure 6-22**

FIGURE 6-22

Cómo se ven las listas de tareas de Markdown.

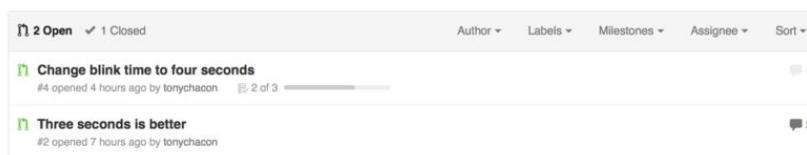
The screenshot shows a comment from user tonychacon:

tonychacon commented 4 hours ago
This PR replaces #2 as a rebased branch instead.

Write the code
 Write all the tests
 Document the code

Esto se suele usar en Pull Requests para indicar qué cosas hay que hacer en la rama antes de considerar que el Pull Request está listo para fusionarse. La parte realmente interesante es que puedes pulsar los marcadores para actualizar el comentario indicando qué tareas se finalizaron, sin necesidad de editar el texto markdown del mismo.

Además, GitHub mostrará esas listas de tareas como metadatos de las páginas que las muestran. Por ejemplo, si tienes un Pull Request con tareas y miras la página resumen de todos los Pull Request, podrás ver cuánto trabajo queda pendiente. Esto ayuda a la gente a dividir los Pull Requests en subtareas y ayuda a otras personas a seguir la evolución de la rama. Se puede ver un ejemplo de esto en **Figure 6-23**.

**FIGURE 6-23**

Resumen de lista de tareas en la lista de PR.

Esto es increíblemente útil cuando se abre un Pull Request al principio y se quiere usar para seguir el progreso de desarrollo de la característica.

Fragments of código

También se pueden añadir fragmentos de código a los comentarios. Esto resulta útil para mostrar algo que te gustaría probar antes de ponerlo en un commit de tu rama. Esto también se suele usar para añadir ejemplos de código que no funciona u otros asuntos.

Para añadir un fragmento de código, lo tienes que encerrar entre los símbolos del siguiente ejemplo.

```
```java
for(int i=0 ; i < 5 ; i++)
{
 System.out.println("i is : " + i);
}
```

```

Si añades junto a los símbolos el nombre de un lenguaje de programación, como hacemos aquí con *java*, GitHub intentará hacer el resultado de la sintaxis del lenguaje en el fragmento. En el caso anterior, quedaría con el aspecto de **Figure 6-24**.

FIGURE 6-24

Cómo se ve el fragmento de código.



Citas

Si estás respondiendo a un comentario grande, pero solo a una pequeña parte, puedes seleccionar la parte que te interesa y citarlo, para lo que precedes cada línea citada del símbolo >. Esto es tan útil que hay un atajo de teclado para hacerlo: si seleccionas el texto al que quieras contestar y pulsas la tecla r, creará una cita con ese texto en la caja del comentario.

Un ejemplo de cita:

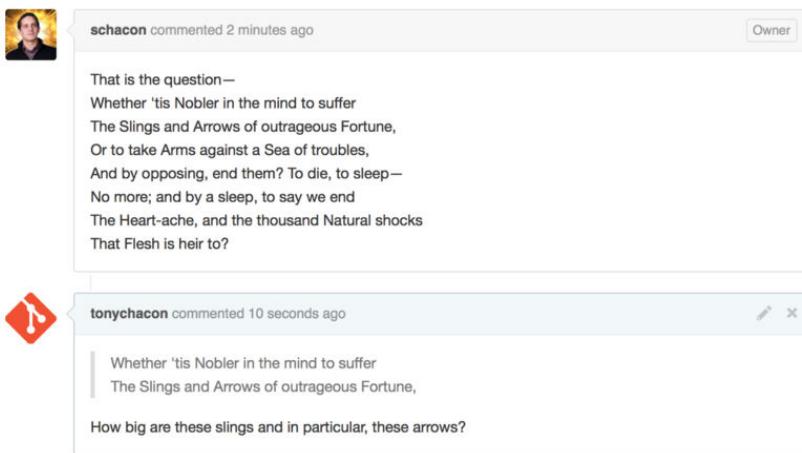
> Whether 'tis Nobler in the mind to suffer
> The Slings and Arrows of outrageous Fortune,

How big are these slings and in particular, these arrows?

Una vez introducida, el comentario se vería como en **Figure 6-25**.

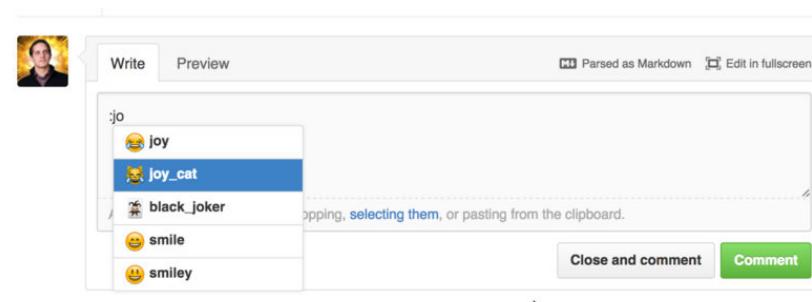
FIGURE 6-25

Rendered quoting example.



Emojis (emoticonos)

Finalmente, también puedes usar emoji (emoticonos) en tus comentarios. Se utiliza mucho en las discusiones de las incidencias y Pull Requests de GitHub. Incluso tenemos un asistente de emoji: si escribes un comentario y tecleas el carácter :, verás cómo aparecen iconos para ayudarte a completar el que quieras poner.

**FIGURE 6-26**

Emoji auto-completando emoji.

Los emoticonos son de la forma :nombre: en cualquier punto del comentario. Por ejemplo, podrás escribir algo como esto:

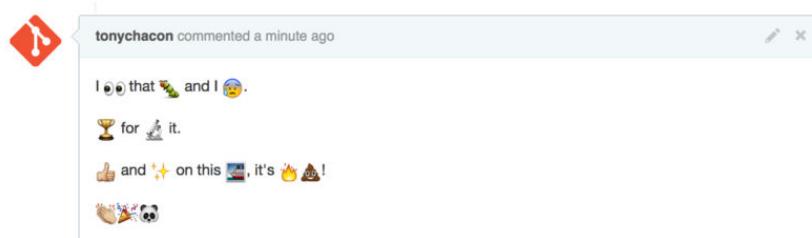
```
I :eyes: that :bug: and I :cold_sweat:.

:trophy: for :microscope: it.

:+1: and :sparkles: on this :ship:, it's :fire::poop:!

:clap::tada::panda_face:
```

Al introducir el comentario, se mostraría como **Figure 6-27**.

**FIGURE 6-27**

Comentando con muchos emoji.

No es que sean especialmente útiles, pero añaden un elemento de gracia y emoción a un medio en el que de otro modo sería mucho más complicado transmitir las emociones.

Actualmente hay bastantes sitios web que usan los emoticonos. Hay una referencia interesante para encontrar el emoji que necesitas en cada momento:

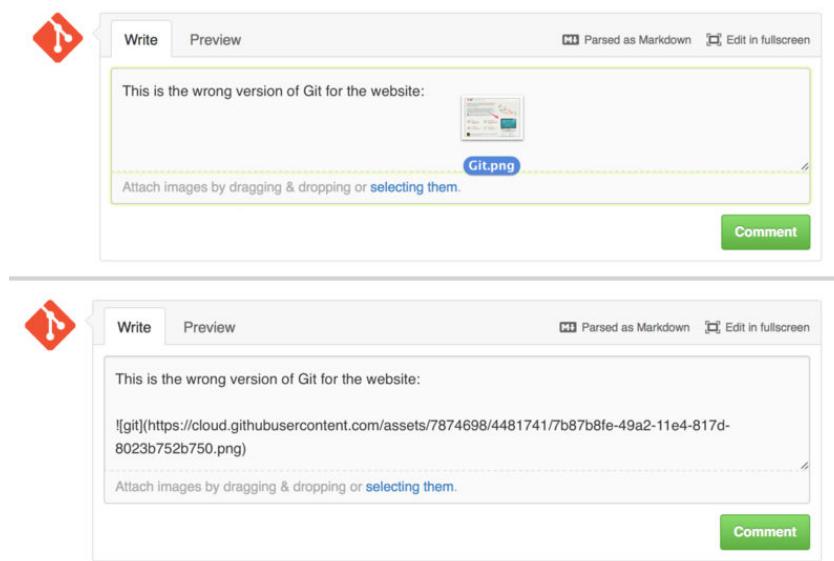
<http://www.emoji-cheat-sheet.com>

Imágenes

Esto no es técnicamente parte de las mejoras a Markdown de GitHub, pero es increíblemente útil. En adición a añadir enlaces con imágenes en el formato Markdown a los comentarios, GitHub permite arrastrar y soltar imágenes en las áreas de texto para insertarlas.

FIGURE 6-28

Arrastrar y soltar imágenes para subirlas.



Si vuelves a **Figure 6-18**, verás una pequeña nota sobre el área de texto “Parsed as Markdown”. Si pulsas ahí te dará una lista completa de cosas que puedes hacer con el formato Markdown de GitHub.

Mantenimiento de un proyecto

Ahora que ya sabes cómo ayudar a un proyecto, veamos el otro lado: cómo puedes crear, administrar y mantener tu propio proyecto.

Creación de un repositorio

Vamos a crear un nuevo repositorio para compartir nuestro código en él. Comienza pulsando el botón “New repository” en el lado derecho de tu página principal, o bien desde el botón + en la barra de botones cercano a tu nombre de usuario, tal como se ve en **Figure 6-30**.

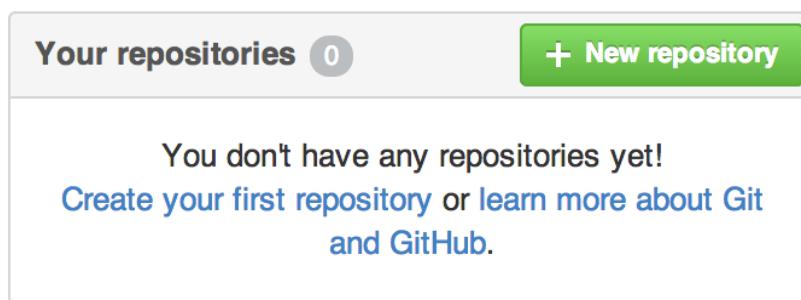


FIGURE 6-29

La zona “Your repositories”.

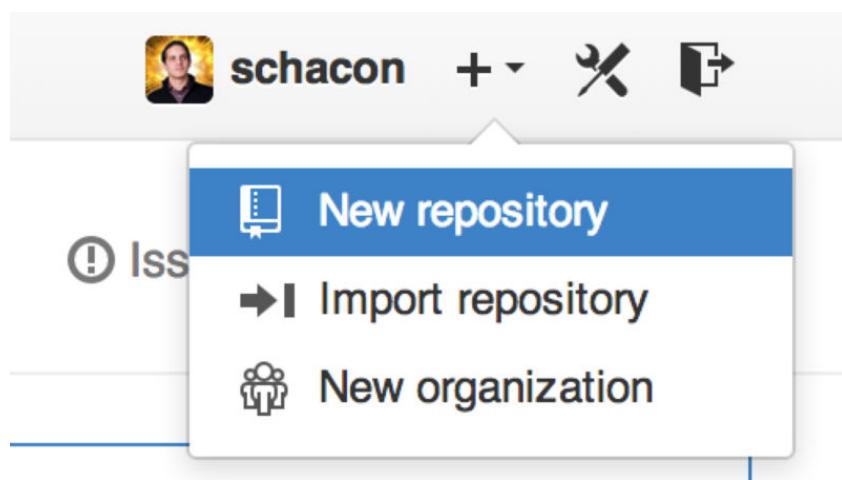


FIGURE 6-30

Desplegable “New repository”.

Esto te llevará al formulario para crear un nuevo repositorio:

FIGURE 6-31

Formulario para crear repositorio.

The screenshot shows the GitHub interface for creating a new repository. At the top, it says "Owner" (ben) and "Repository name" (iOSApp). Below that is a note: "Great repository names are short and memorable. Need inspiration? How about [drunken-dubstep](#)." Under "Description (optional)", there is a text input field containing "iOS project for our mobile group". There are two radio buttons for visibility: "Public" (selected) and "Private". Below these, there is a checkbox for "Initialize this repository with a README", which includes a note: "This will allow you to `git clone` the repository immediately. Skip this step if you have already run `git init` locally." At the bottom, there are buttons for "Add .gitignore: None" and "Add a license: None". A large green "Create repository" button is at the very bottom.

Todo lo que tienes que hacer aquí es darle un nombre al proyecto; el resto de campos es totalmente opcional. Por ahora, pulsa en el botón “Create Repository” y listo: se habrá creado el repositorio en GitHub, con el nombre <usuario>/<proyecto>

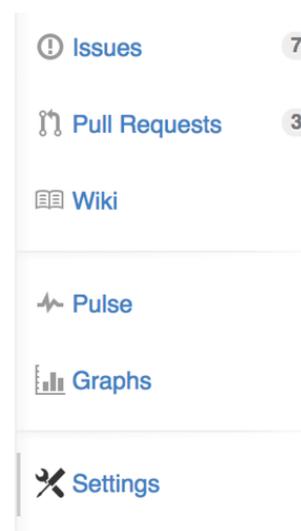
Dado que no tiene todavía contenido, GitHub te mostrará instrucciones para crear el repositorio Git, o para conectarlo a un proyecto Git existente. No entraremos aquí en esto; si necesitas refresharlo, revisa el capítulo **Chapter 2**.

Ahora que el proyecto está alojado en GitHub, puedes dar la URL a cualquiera con quien quieras compartirlo. Cada proyecto en GitHub es accesible mediante HTTP como <https://github.com/<usuario>/<proyecto>>, y también con SSH con la dirección `git@github.com:<usuario>/<proyecto>`. Git puede obtener y enviar cambios en ambas URL, ya que tienen control de acceso basado en las credenciales del usuario.

Suele ser preferible compartir la URL de tipo HTTP de los proyectos públicos, puesto que así el usuario no necesitará una cuenta GitHub para clonar el proyecto. Si das la dirección SSH, los usuarios necesitarán una cuenta GitHub y subir una clave SSH para acceder. Además, la URL HTTP es exactamente la misma que usamos para ver la página web del proyecto.

Añadir colaboradores

Si estás trabajando con otras personas y quieres darle acceso de escritura, necesitarás añadirlas como “colaboradores”. Si Ben, Jeff y Louise se crean cuentas en GitHub, y quieres darles acceso de escritura a tu repositorio, los tienes que añadir al proyecto. Al hacerlo le darás permiso de “push”, que significa que tendrán tanto acceso de lectura como de escritura, en el proyecto y en el repositorio Git.

**FIGURE 6-32**

Enlace a ajustes del repositorio.

Selecciona “Collaborators” del menú del lado izquierdo. Simplemente, teclea el usuario en la caja, y pulsa en “Add collaborator.” Puedes repetir esto las veces que necesites para dar acceso a otras personas. Si necesitas quitar un acceso, pulsa en la “X” del lado derecho del usuario.

| Collaborators | | Full access to the repository |
|---------------|-----------------------------------|-------------------------------|
| | Ben Straub
ben | X |
| | Jeff King
peff | X |
| | Louise Corrigan
LouiseCorrigan | X |

FIGURE 6-33

Colaboradores del repositorio.

Gestión de los Pull Requests

Ahora que tienes un proyecto con algo de código, y probablemente algunos colaboradores con acceso de escritura, veamos qué pasa cuando alguien te hace un Pull Request.

Los Pull Requests pueden venir de una rama en una bifurcación del repositorio, o pueden venir de una rama pero del mismo repositorio. La única diferencia es que, en el primer, caso procede de gente que no tiene acceso de escritura a tu proyecto y quiere integrar en el tuyo cambios interesantes, mientras que en el segundo caso procede de gente con acceso al repositorio.

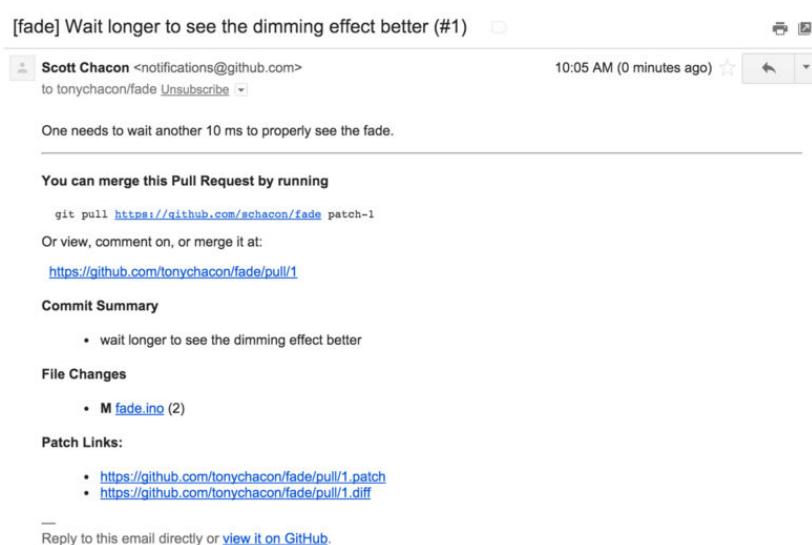
En los siguientes ejemplos, supondremos que eres “tonychacon” y has creado un nuevo proyecto para Arduino llamado “fade”.

NOTIFICACIONES POR CORREO ELECTRÓNICO

Cuando alguien realiza un cambio en el código y te crea un Pull Request, debes recibir una notificación por correo electrónico avisándote, con un aspecto similar a **Figure 6-34**.

FIGURE 6-34

Notificación por correo de nuevo Pull Request.



Hay algunas cosas a destacar en este correo. En primer lugar, te dará un pequeño diffstat (es decir, una lista de ficheros cambiados y en qué medida). Ade-

más, trae un enlace al Pull Request y algunas URL que puedes usar desde la línea de comandos.

Si observas la línea que dice `git pull <url> patch-1`, es una forma simple de fusionar una rama remota sin tener que añadirla localmente. Lo vimos esto rápidamente en “**Recuperando ramas remotas**”. Si lo deseas, puedes crear y cambiar a una rama y luego ejecutar el comando para fusionar los cambios del Pull Request.

Las otras URL interesantes son las de `.diff` y `.patch`, que como su nombre indican, proporcionan diff unificados y formatos de parche del Pull Request. Técnicamente, podrías fusionar con algo como:

```
$ curl http://github.com/tonychacon/fade/pull/1.patch | git am
```

COLABORACIÓN EN EL PULL REQUEST

Como hemos visto en “**El Flujo de Trabajo en GitHub**”, puedes participar en una discusión con la persona que generó el Pull Request. Puedes comentar líneas concretas de código, comentar commits completos o comentar el Pull Request en sí mismo, utilizando donde quieras el formato Markdown.

Cada vez que alguien comenta, recibirás nuevas notificaciones por correo, lo que te permite vigilar todo lo que pasa. Cada correo tendrá un enlace a la actividad que ha tenido lugar, y además puedes responder al comentario simplemente contestando al correo.

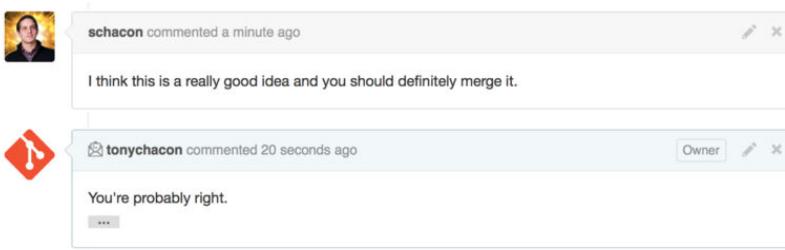


FIGURE 6-35

Las respuestas a correos se incluyen en el hilo de discusión.

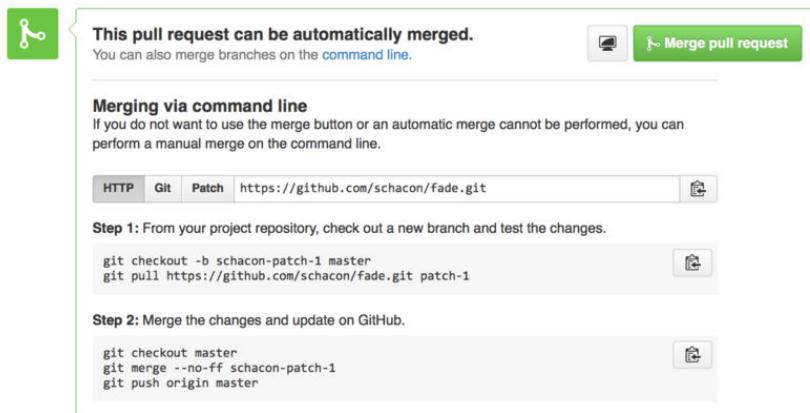
Una vez que el código está como quieras y quieres fusionarlo, puedes copiar el código y fusionarlo localmente, mediante la sintaxis ya conocida de `git pull <url> <branch>`, o bien añadiendo el fork como nuevo remoto, bajándotelo y luego fusionándolo.

Si la fusión es trivial, también puedes pulsar el botón “Merge” en GitHub. Esto realizará una fusión “sin avance rápido”, creando un commit de fusión inclu-

so si era posible una fusión con avance rápido. Esto significa que cada vez que pulses el botón Merge, se creará un commit de fusión. Como verás en **Figure 6-36**, GitHub te da toda esta información si pulsas el el enlace de ayuda.

FIGURE 6-36

Botón Merge e instrucciones para fusionar manualmente un Pull Request.



Si decides que no quieres fusionar, también puedes cerrar el Pull Request y la persona que lo creó será notificada.

REFERENCIAS DE PULL REQUEST

Si tienes muchos Pull Request y no quieres añadir un montón de remotos o hacer muchos cada vez, hay un pequeño truco que GitHub te permite. Es un poco avanzado y lo veremos en detalle después en “**The Refspec**”, pero puede ser bastante útil.

En GitHub tenemos que las ramas de Pull Request son una especie de pseudo-ramas del servidor. De forma predeterminada no las obtendrás cuando hagas un clonado, pero hay una forma algo oscura de acceder a ellos.

Para demostrarlo, usaremos un comando de bajo nivel (conocido como de “fontanería”, sabremos más sobre esto en “**Plumbing and Porcelain**”) llamado `ls-remote`. Este comando no se suele usar en el día a día de Git pero es útil para ver las referencias presentes en el servidor.

Si ejecutamos este comando sobre el repositorio “blink” que hemos estado usando antes, obtendremos una lista de ramas, etiquetas y otras referencias del repositorio.

```
$ git ls-remote https://github.com/schacon/blink
10d539600d86723087810ec636870a504f4fee4d          HEAD
```

```

10d539600d86723087810ec636870a504f4fee4d      refs/heads/master
6a83107c62950be9453aac297bb0193fd743cd6e      refs/pull/1/head
afe83c2d1a70674c9505cc1d8b7d380d5e076ed3      refs/pull/1/merge
3c8d735ee16296c242be7a9742ebfb2665adec1      refs/pull/2/head
15c9f4f80973a2758462ab2066b6ad9fe8dcf03d      refs/pull/2/merge
a5a7751a33b7e86c5e9bb07b26001bb17d775d1a      refs/pull/4/head
31a45fc257e8433c8d8804e3e848cf61c9d3166c      refs/pull/4/merge

```

Por supuesto, si estás en tu repositorio y tecleas `git ls-remote origin` podrás ver algo similar pero para el remoto etiquetado como `origin`.

Si el repositorio está en GitHub y tienes Pull Requests abiertos, tendrás estas referencias con el prefijo `refs/pull`. Básicamente, son ramas, pero ya que no están bajo `refs/heads/`, no las obtendrás normalmente cuando clonas o te bajas el repositorio del servidor, ya que el proceso de obtención las ignora.

Hay dos referencias por cada Pull Request, la que termina en `/head` apunta exactamente al último commit de la rama del Pull Request. Así si alguien abre un Pull Request en el repositorio y su rama se llama `bug-fix` apuntando al commit `a5a775`, en nuestro repositorio no tendremos una rama `bug-fix` (puesto que está en el fork) pero tendremos el `pull/<pr#>/head` apuntando a `a5a775`. Esto significa que podemos obtener fácilmente cada Pull Request sin tener que añadir un montón de remotos.

Ahora puedes obtenerlo directamente.

```

$ git fetch origin refs/pull/958/head
From https://github.com/libgit2/libgit2
 * branch                  refs/pull/958/head -> FETCH_HEAD

```

Esto dice a Git, “Conecta al remoto `origin` y descarga la referencia llamada `refs/pull/958/head`.” Git obedece y descarga todo lo necesario para construir esa referencia, y deja un puntero al commit que quieras bajo `.git/FETCH_HEAD`. Puedes realizar operaciones como `git merge FETCH_HEAD` aunque el mensaje del commit será un poco confuso. Además, si estás revisando un montón de pull requests, se convertirá en algo tedioso.

Hay también una forma de obtener *todos* los pull requests, y mantenerlos actualizados cada vez que conectas al remoto. Para ello abre el fichero `.git/config` y busca la línea `origin`. Será similar a esto:

```

[remote "origin"]
  url = https://github.com/libgit2/libgit2
  fetch = +refs/heads/*:refs/remotes/origin/*

```

La línea que comienza con `fetch =` es un “refspec.” Es una forma de mapear nombres del remoto con nombres de tu copia local. Este caso concreto dice a Git, que “las cosas en el remoto bajo `refs/heads` deben ir en mi repositorio bajo `refs/remotes/origin`.” Puedes modificar esta sección añadiendo otra refspect:

```
[remote "origin"]
  url = https://github.com/libgit2/libgit2.git
  fetch = +refs/heads/*:refs/remotes/origin/*
  fetch = +refs/pull/*/head:refs/remotes/origin/pr/*
```

Con esta última línea decimos a Git, “Todas las referencias del tipo `refs/pull/123/head` deben guardarse localmente como `refs/remotes/origin/pr/123`.” Ahora, si guardas el fichero y ejecutas un `git fetch` tendremos:

```
$ git fetch
# ...
* [new ref]      refs/pull/1/head -> origin/pr/1
* [new ref]      refs/pull/2/head -> origin/pr/2
* [new ref]      refs/pull/4/head -> origin/pr/4
# ...
```

Ya tienes todos los pull request en local de forma parecida a las ramas; son solo-lectura y se actualizan cada vez que haces un `fetch`. Pero hace muy fácil probar el código de un pull request en local:

```
$ git checkout pr/2
Checking out files: 100% (3769/3769), done.
Branch pr/2 set up to track remote branch pr/2 from origin.
Switched to a new branch 'pr/2'
```

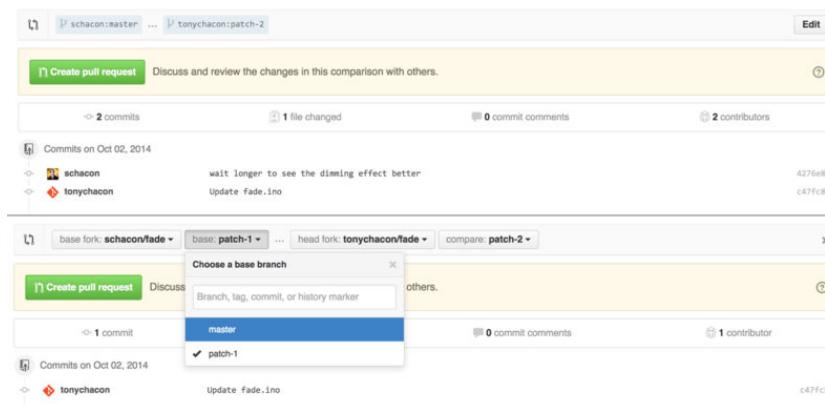
La referencia `refs/pull/#/merge` de GitHub representa el commit que resultaría si pulsamos el botón “merge”. Esto te permite probar la fusión del pull request sin llegar a pulsar dicho botón.

PULL REQUESTS SOBRE PULL REQUESTS

No solamente se puede abrir Pull Requests en la rama `master`, también se pueden abrir sobre cualquier rama de la red. De hecho, puedes poner como objetivo otro Pull Request.

Si ves que un Pull Request va en la buena dirección y tienes una idea para hacer un cambio que depende de él, o bien no estás seguro de que sea una buena idea, o no tienes acceso de escritura en la rama objetivo, puedes abrir un Pull Request directamente.

Cuando vas a abrir el Pull Request, hay una caja en la parte superior de la página que especifica qué rama quieras usar y desde qué rama quieres hacer la petición. Si pulsas el botón “Edit” en el lado derecho de la caja, puedes cambiar no solo las ramas sino también la bifurcación.

**FIGURE 6-37**

Cambio manual de la rama o del fork en un pull request.

Aquí puedes fácilmente especificar la fusión de tu nueva rama en otro Pull Request o en otra bifurcación del proyecto.

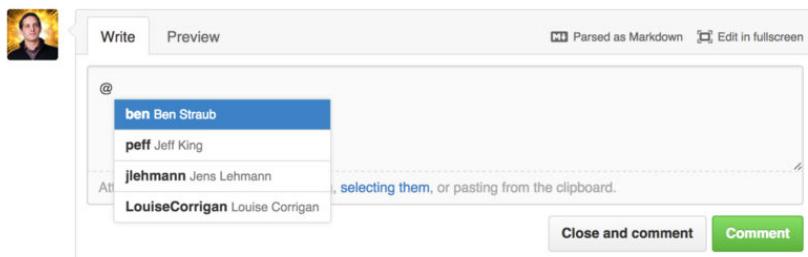
Menciones y notificaciones

GitHub tiene un sistema de notificaciones que resulta útil cuando necesitas pedir ayuda o necesitas la opinión de otros usuarios o equipos concretos.

En cualquier comentario, si comienzas una palabra comenzando con el carácter @, intentará auto-completar nombres de usuario de personas que sean colaboradores o responsables en el proyecto.

FIGURE 6-38

Empieza tecleando @ para mencionar a alguien.



También puedes mencionar a un usuario que no esté en la lista desplegable, pero normalmente el autocomplete lo hará más rápido.

Una vez que envías un comentario con mención a un usuario, el usuario citado recibirá una notificación. Es decir, es una forma de implicar más gente en una conversación. Esto es muy común en los Pull Requests para invitar a terceros a que participen en la revisión de una incidencia o un pull request.

Si alguien es mencionado en un Pull request o incidencia, quedará además “suscripto” y recibirá desde este momento las notificaciones que genere su actividad. Del mismo modo, el usuario que crea la incidencia o el Pull request queda automáticamente suscrito para recibir las notificaciones, disponiendo todos de un botón “Unsubscribe” para dejar de recibirlas.

FIGURE 6-39

Quitar suscripción de un pull request o incidencia.

Notifications

Unsubscribe

You're receiving notifications because you commented.

PÁGINA DE NOTIFICACIONES

Cuando decimos “notificaciones”, nos referimos a una forma por la que GitHub intenta contactar contigo cuando tienen lugar eventos, y éstas pueden ser configuradas de diferentes formas. Si te vas al enlace “Notification center” de la página de ajustes, verás las diferentes opciones disponibles.

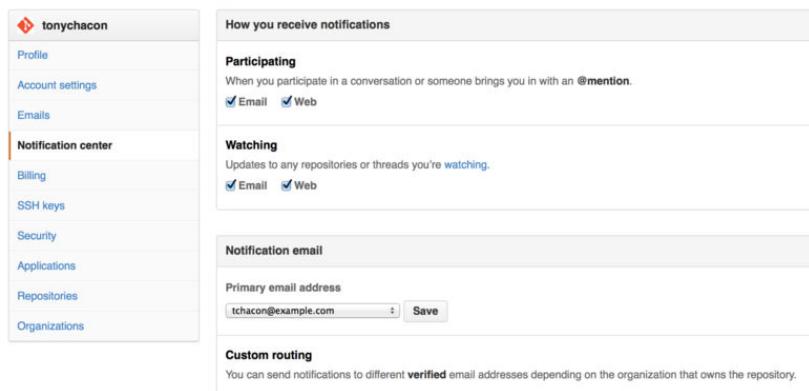


FIGURE 6-40

Opciones de
Notification center.

Para cada tipo, puedes elegir tener notificaciones de “Email” o de “Web”, y puedes elegir tener una de ellas, ambas o ninguna.

Notificaciones Web

Las notificaciones web se muestran en la página de Github. Si las tienes activas verás un pequeño punto azul sobre el ícono de notificaciones en la parte superior de la pantalla, en **Figure 6-41**.

FIGURE 6-41

Centro de notificaciones.

The screenshot shows the GitHub Notifications center. At the top, there's a search bar labeled 'Search GitHub' and navigation links for 'Explore', 'Gist', 'Blog', and 'Help'. On the right, it shows the user 'tonychacon' with a profile icon and a notification badge. A message says 'You have unread notifications' with a 'Mark all as read' button. Below this, there are two main sections: 'Notifications' and 'Watching'.

- Notifications:** Shows 4 unread notifications. One from 'mycorp/project1' about a 'SF Corporate Housing Search' made an hour ago. Another from 'git/git-scm.com' about the 'Front Page' updated 3 hours ago. Two notifications from 'schacon/blink': one about 'To Be or Not To Be' and another about 'Three seconds is better', both updated 5 days ago. Each notification has a checkmark icon indicating it's been read.
- Watching:** Shows 3 notifications. One from 'schacon/blink' and two from 'mycorp/project1'.

Si pulsas en él, verás una lista de todos los elementos sobre los que se te notifica, agrupados por proyecto. Puedes filtrar para un proyecto específico pulsando en el nombre en el lado izquierdo. También puedes reconocer (marcar como leída) una notificación pulsando en el ícono de check en una notificación, o reconocerlas *todas* pulsando en el ícono de check de todo el grupo. Hay también un botón “mute” para silenciarlas, que puedes pulsar para no recibir nuevas notificaciones de ese elemento en el futuro.

Todas estas utilidades son útiles para manejar un gran número de notificaciones. Muchos usuarios avanzados de GitHub suelen desactivar las notificaciones por correo y manejarlas todas mediante esta pantalla.

Notificaciones por correo

Las notificaciones por correo electrónico son la otra manera de gestionar notificaciones con GitHub. Si las tienes activa, recibirás los correos de cada notificación. Vimos ya algún ejemplo en [Figure 6-13](#) y [Figure 6-34](#). Los correos también serán agrupados correctamente en conversaciones, con lo que estará bien que uses un cliente de correo que maneje las conversaciones.

En las cabeceras de estos correos se incluyen también algunos metadatos, que serán útiles para crear filtros y reglas adecuados.

Por ejemplo, si miramos las cabeceras de los correos enviados a Tony en el correo visto en [Figure 6-34](#), veremos que se envió la siguiente información:

```
To: tonychacon/fade <fade@noreply.github.com>
Message-ID: <tonychacon/fade/pull/1@github.com>
Subject: [fade] Wait longer to see the dimming effect better (#1)
X-GitHub-Recipient: tonychacon
List-ID: tonychacon/fade <fade.tonychacon.github.com>
List-Archive: https://github.com/tonychacon/fade
List-Post: <mailto:reply+i-4XXX@reply.github.com>
```

List-Unsubscribe: <mailto:unsub+i-XXX@reply.github.com>, ...
X-GitHub-Recipient-Address: tchacon@example.com

Vemos en primer lugar que la información de la cabecera `Message-Id` nos da los datos que necesitamos para identificar usuario, proyecto y demás en formato `<usuario>/<proyecto>/<tipo>/<id>`. Si se tratase de una incidencia, la palabra “pull” habría sido reemplazada por “issues”.

Las cabeceras `List-Post` y `List-Unsubscribe` sirven a clientes de correo capaces de interpretarlas ayudarnos a solicitar dejar de recibir nuevas notificaciones de ese tema. Esto es similar a pulsar el botón “mute” que vimos en la versión web, o en “Unsubscribe” en la página de la incidencia o el Pull Request.

También merece la pena señalar que si tienes activadas las notificaciones tanto en la web como por correo, y marcas como leído el correo, en la web también se marcará como leído siempre que permitas las imágenes en el cliente de correo.

Ficheros especiales

Hay dos ficheros especiales que GitHub detecta y maneja si están presentes en el repositorio.

README

En primer lugar tenemos el fichero `README`, que puede estar en varios formatos. Puede estar con el nombre `README`, `README.md`, `README.asciidoc` y alguno más. Cuando GitHub detecta la presencia del proyecto, lo muestra en la página principal, con el *renderizado* que corresponda a su formato.

En muchos casos este fichero se usa para mostrar información relevante a cualquiera que sea nuevo en el proyecto o repositorio. Esto incluye normalmente cosas como:

- Para qué es el proyecto
- Cómo se configura y se instala
- Ejemplo de uso
- Licencia del código del proyecto
- Cómo participar en su desarrollo

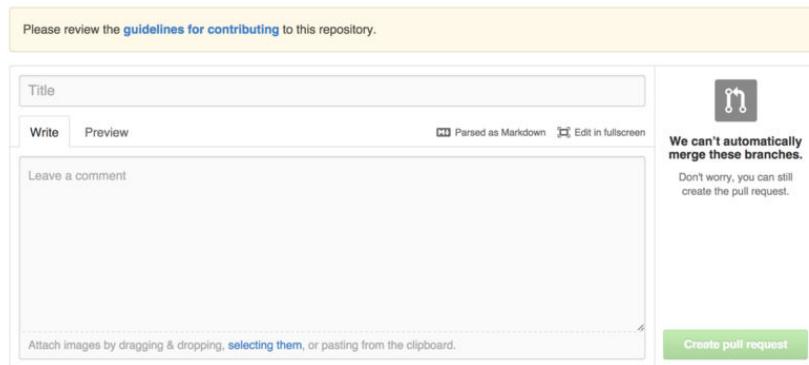
Puesto que GitHub hace un renderizado del fichero, puedes incluir imágenes o enlaces en él para facilitar su comprensión.

CONTRIBUTING

El otro fichero que GitHub reconoce es el fichero CONTRIBUTING. Si tienes un fichero con ese nombre y cualquier extensión, GitHub mostrará algo como **Figure 6-42** cuando se intente abrir un Pull Request.

FIGURE 6-42

Apertura de un Pull Request cuando existe el fichero CONTRIBUTING.



La idea es que indiques cosas a considerar a la hora de recibir un Pull Request. La gente lo debe leer a modo de guía sobre cómo abrir la petición.

Administración del proyecto

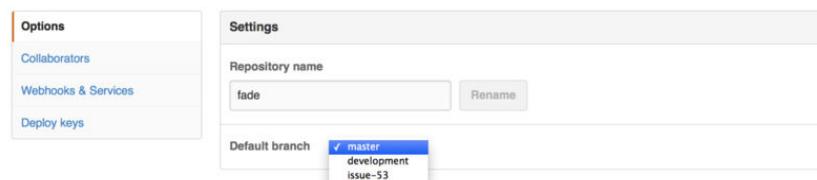
Por lo general, no hay muchas cosas que administrar en un proyecto concreto, pero sí un par de cosas que pueden ser interesantes.

CAMBIAR LA RAMA PREDETERMINADA

Si usas como rama predeterminada una que no sea “master”, por ejemplo para que sea objetivo de los Pull Requests, puedes cambiarla en las opciones de configuración del repositorio, en donde pone “Options”.

FIGURE 6-43

Cambio de la rama predeterminada del proyecto.



Simplemente cambia la rama predeterminada en la lista desplegable, y ésta será la elegida para la mayoría de las operaciones, así mismo será la que sea visible al principio (“checked-out”) cuando alguien clona el repositorio.

TRANSFERENCIA DE UN PROYECTO

Si quieras transferir la propiedad de un proyecto a otro usuario u organización de GitHub, hay una opción para ello al final de “Options” llamada “Transfer ownership”.

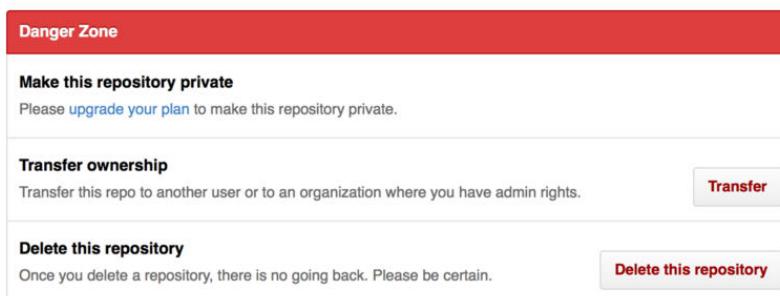


FIGURE 6-44

Transferir propiedad de un proyecto.

Esto es útil si vas a abandonar el proyecto y quieras que alguien continúe, o bien se ha vuelto muy grande y prefieres que se gestione desde una organización.

Esta transferencia, supone un cambio de URL. Para evitar que nadie se pierda, genera una redirección web en la URL antigua. Esta redirección funciona también con las operaciones de clonado o de copia desde Git.

Gestión de una organización

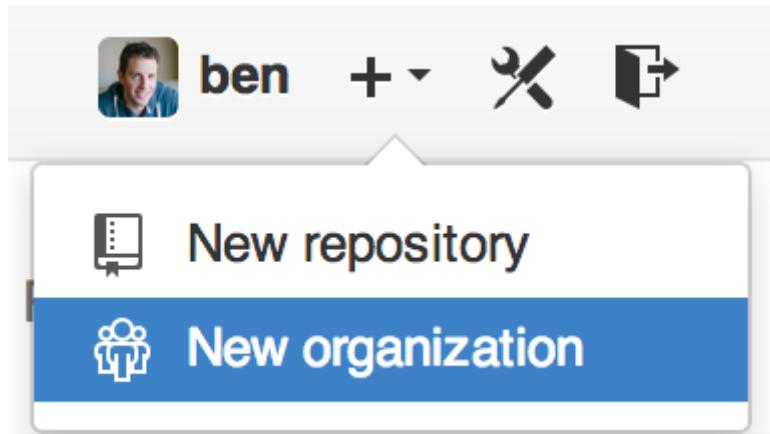
Además de las cuentas de usuario, GitHub tiene Organizaciones. Al igual que las cuentas de usuario, las cuentas de organización tienen un espacio donde se guardarán los proyectos, pero en otras cosas son diferentes. Estas cuentas representan un grupo de gente que comparte la propiedad de los proyectos, y además se pueden gestionar estos miembros en subgrupos. Normalmente, estas cuentas se usan en equipos de desarrollo de código abierto (por ejemplo, un grupo para “perl” o para “rails”) o empresas (como sería ``google” o “twitter”).

Conceptos básicos

Crear una organización es muy fácil: simplemente pulsa en el icono “+” en el lado superior derecho y selecciona “New organization”.

FIGURE 6-45

El menú “New organization”.



En primer lugar tienes que decidir el nombre de la organización y una dirección de correo que será el punto principal de contacto del grupo. A continuación puedes invitar a otros usuarios a que se unan como co-propietarios de la cuenta.

Sigue estos pasos y serás propietario de un grupo nuevo. Al igual que las cuentas personales, las organizaciones son gratuitas siempre que los repositorios sean de código abierto (y por tanto, públicos).

Como propietario de la organización, cuando bifurcas un repositorio podrás hacerlo a tu elección en el espacio de la organización. Cuando creas nuevos repositorios puedes también elegir el espacio donde se crearán: la organización o tu cuenta personal. Automáticamente, además, quedarás como vigilante (watcher) de los repositorios que crees en la organización.

Al igual que en “**Tu ícono**”, puedes subir un ícono para personalizar un poco la organización, que aparecerá entre otros sitios en la página principal de la misma, que lista todos los repositorios y puede ser vista por cualquiera.

Vamos a ver algunas cosas que son diferentes cuando se hacen con una cuenta de organización.

Equipos

Las organizaciones se asocian con individuos mediante los equipos, que son simplemente agrupaciones de cuentas de usuario y repositorios dentro de la organización, y qué accesos tienen esas personas sobre cada repositorio.

Por ejemplo, si tu empresa tiene tres repositorios: `frontend`, `backend` y `deployscripts`; y quieres que los desarrolladores de web tengan acceso a `frontend` y tal vez a `backend`, y las personas de operaciones tengan acceso a `backend` y `deployscripts`. Los equipos hacen fácil esta organización, sin tener que gestionar los colaboradores en cada repositorio individual.

La página de la organización te mostrará un panel simple con todos los repositorios, usuarios y equipos que se encuentran en ella.

The screenshot shows the GitHub organization page for 'chaconcorp'. At the top, there's a logo, the organization name 'chaconcorp', and a search bar labeled 'Find a repository...'. Below this, there are three repository cards: 'deployscripts' (scripts for deployment, updated 16 hours ago), 'backend' (Backend Code, updated 16 hours ago), and 'frontend' (Frontend Code, updated 16 hours ago). To the right, there are two main sections: 'People' and 'Teams'. The 'People' section lists three members: 'dragonchacon' (Dragon Chacon), 'schacon' (Scott Chacon), and 'tonychacon' (Tony Chacon). Each member has a small profile picture and a link to their profile. The 'Teams' section shows three teams: 'Owners' (1 member · 3 repositories), 'Frontend Developers' (2 members · 2 repositories), and 'Ops' (3 members · 1 repository). Each team has a 'Jump to a team' button and a 'Create new team' button at the bottom.

FIGURE 6-46

Página de la organización.

Para gestionar tus equipos, puedes pulsar en la barra “Teams” del lado derecho en la página **Figure 6-46**. Esto te llevará a una página con la que puedes añadir los miembros del equipo, añadir repositorios al equipo o gestionar los ajustes y niveles de acceso del mismo. Cada equipo puede tener acceso sólo lectura, de escritura o administrativo al repositorio. Puedes cambiar el nivel pulsando en el botón “Settings” en **Figure 6-47**.

FIGURE 6-47

Página de equipos.

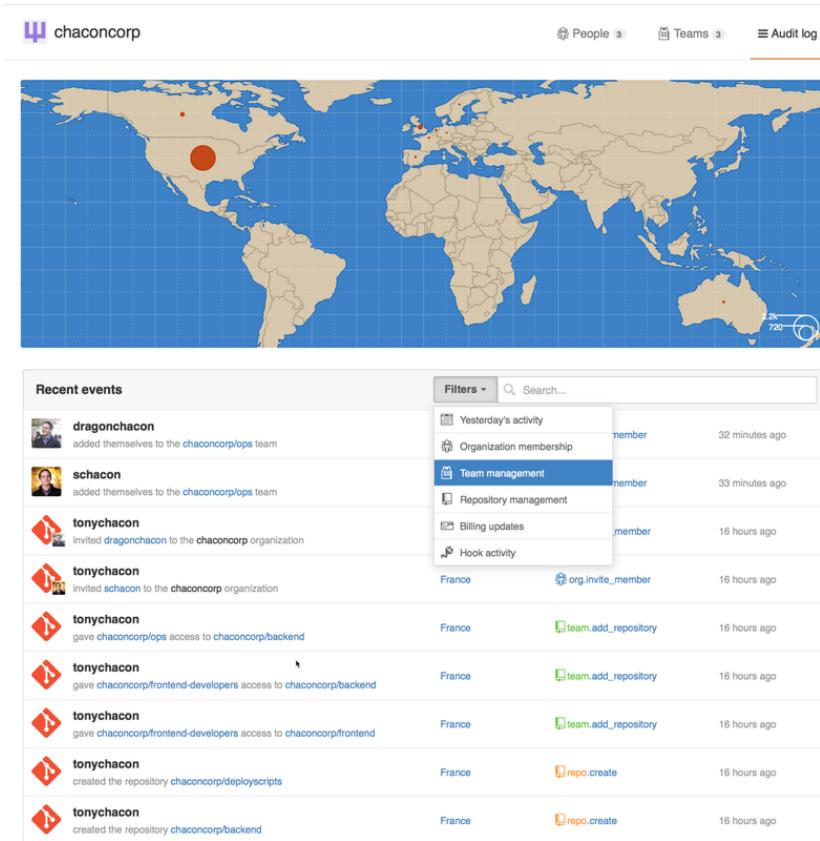
The screenshot shows the GitHub interface for managing teams. At the top, there's a navigation bar with 'People', 'Teams', and 'Audit log'. Below that, the 'Frontend Developers' team page is displayed. The team has 2 members and 2 repositories. Two members are listed: 'tonychacon' (Tony Chacon) and 'schacon' (Scott Chacon). Each member has a 'Remove' button next to their name. A 'Members' tab is selected, and a 'Repositories' tab is also present. A button to 'Invite or add users to team' is located at the top right. A note below the team summary states: 'This team grants **Admin** access: members can read from, push to, and add collaborators to the team's repositories.'

Cuando invitas a alguien a un equipo, recibirá un correo con una invitación. Además, hay menciones de equipo (por ejemplo, @acmecorp/frontend) que servirán para que todos los miembros de ese equipo sean suscritos al hilo. Esto resulta útil si quieras involucrar a un equipo en algo al no tener claro a quién concreto preguntar.

Un usuario puede pertenecer a cuantos equipos desee, por lo que no uses equipos solamente para temas de control de acceso a repositorios, sino que puedes usarlo para formar equipos especializados y dispares como ux, css, refactoring, legal, etc.

Auditorías

Las organizaciones pueden también dar a los propietarios acceso a toda la información sobre la misma. Puedes incluso ir a la opción *Audit Log* y ver los eventos que han sucedido, quién hizo qué y dónde.

**FIGURE 6-48***Log de auditoría.*

También puedes filtrar por tipo de evento, por lugares o por personas concretas.

Scripting en GitHub

Ya conocemos casi todas las características y modos de trabajo de GitHub. Sin embargo, cualquier grupo o proyecto medianamente grande necesitará personalizar o integrar GitHub con servicios externos.

Por suerte para nosotros, GitHub es bastante *hackeable* en muchos sentidos. En esta sección veremos cómo se usan los *enganches* (hooks) de GitHub y las API para conseguir hacer lo que queremos.

Enganches

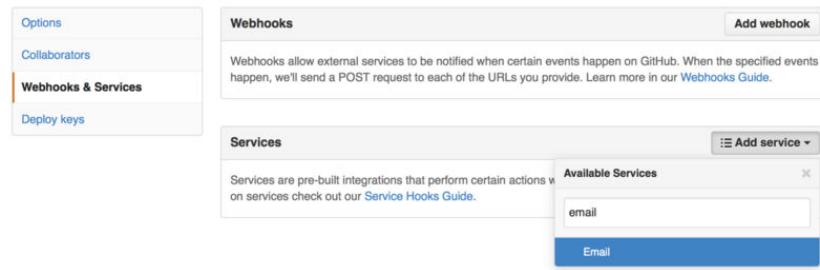
Las secciones Hooks y Services de la página de administración del repositorio en Github es la forma más simple de hacer que GitHub interactúe con sistemas externos.

SERVICIOS

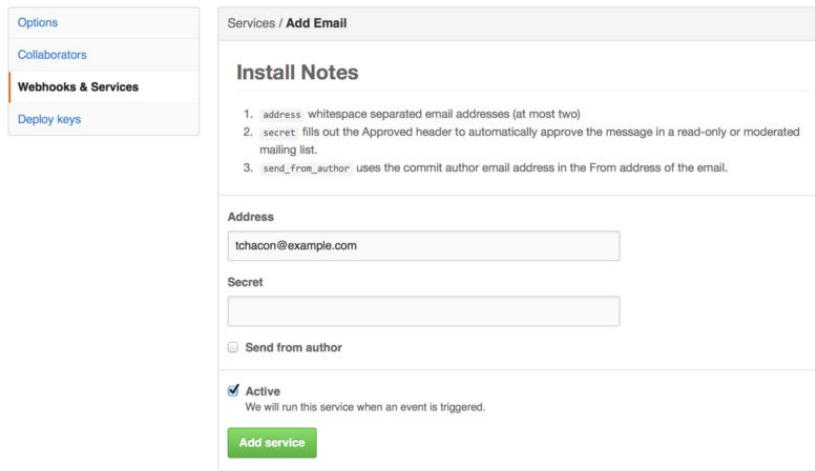
En primer lugar, echaremos un ojo a los Servicios. Ambos, enganches y servicios pueden configurarse desde la sección Settings del repositorio, el mismo sitio donde vimos que podíamos añadir colaboradores al proyecto o cambiar la rama predeterminada. Bajo la opción “Webhooks and Services” veremos algo similar a **Figure 6-49**.

FIGURE 6-49

Sección Services and Hooks.



Hay docenas de servicios que podemos elegir, muchos de ellos para integrarse en otros sistemas de código abierto o comerciales. Muchos son servicios de integración continua, gestores de incidencias y fallos, salas de charla y sistemas de documentación. Veremos cómo levantar un servicio sencillo, el enganche con el correo electrónico. Si elegimos “email” en la opción “Add Service” veremos una pantalla de configuración similar a **Figure 6-50**.

**FIGURE 6-50**

Configuración de servicio de correo.

En este caso, si pulsamos en el botón “Add service”, la dirección de correo especificada recibirá un correo cada vez que alguien envía cambios (push) al repositorio. Los servicios pueden dispararse con muchos otros tipos de eventos, aunque la mayoría solo se usan para los eventos de envío de cambios (push) y hacer algo con los datos del mismo.

Si quieras integrar algún sistema concreto con GitHub, debes mirar si hay algún servicio de integración ya creado. Por ejemplo, si usas Jenkins para ejecutar pruebas de tu código, puedes activar el servicio de integración de Jenkins que lo disparará cada vez que alguien altera el repositorio.

HOOKS (ENGANCHES)

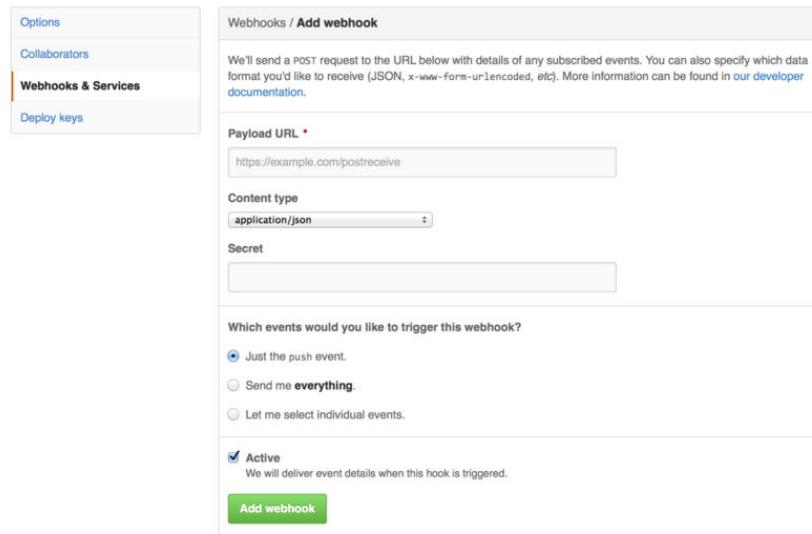
Si necesitas algo más concreto o quieras integrarlo con un servicio o sitio no incluido en la lista, puedes usar el sistema de enganches más genérico. Los enganches de GitHub son bastante simples. Indicas una URL y GitHub enviará una petición HTTP a dicha URL cada vez que suceda el evento que quieras.

Normalmente, esto funcionará si puedes configurar un pequeño servicio web para escuchar las peticiones de GitHub y luego hacer algo con los datos que son enviados.

Para activar un enganche, pulsa en el botón “Add webhook” de **Figure 6-49**. Esto mostrará una página como **Figure 6-51**.

FIGURE 6-51

Configuración de enganches web.



La configuración de un enganche web es bastante simple. Casi siempre basta con incluir una URL y una clave secreta, y pulsar en “Add webhook”. Hay algunas opciones sobre qué eventos quieras que disparen el envío de datos (de forma predeterminada el único evento considerado es el evento push, de cuando alguien sube algo a cualquier rama del repositorio).

Veamos un pequeño ejemplo de servicio web para manejar un enganche web. Usaremos el entorno Sinatra de Ruby puesto que es conciso y podrás entender con facilidad qué estamos haciendo.

Pongamos que queremos recibir un correo cada vez que alguien sube algo a una rama concreta del repositorio, modificando un fichero en particular. Podríamos hacerlo con un código similar a este:

```
require 'sinatra'
require 'json'
require 'mail'

post '/payload' do
  push = JSON.parse(request.body.read) # parse the JSON

  # gather the data we're looking for
  pusher = push["pusher"]["name"]
  branch = push["ref"]

  # get a list of all the files touched
  files = push["commits"].map do |commit|
```

```

    commit['added'] + commit['modified'] + commit['removed']
end
files = files.flatten.uniq

# check for our criteria
if pusher == 'schacon' &&
  branch == 'ref/heads/special-branch' &&
  files.include?('special-file.txt')

  Mail.deliver do
    from      'tchacon@example.com'
    to        'tchacon@example.com'
    subject   'Scott Changed the File'
    body      "ALARM"
  end
end
end

```

Aquí estamos tomando el bloque JSON que GitHub entrega y mirando quién hizo el envío, qué rama se envió y qué ficheros se modificaron en cada commit realizado en este push. Entonces, comprobamos si se cumple nuestro criterio y enviamos un correo si es así.

Para poder probar algo como esto, tienes una consola de desarrollador en la misma pantalla donde configuraste el enganche, donde se puede ver las últimas veces que GitHub ha intentado ejecutar el enganche. Para cada uno, puedes mirar qué información se ha enviado, y si fue recibido correctamente, junto con las cabeceras correspondientes de la petición y de la respuesta. Esto facilita mucho las pruebas de tus enganches.

FIGURE 6-52

Depuración de un web hook.

The screenshot shows the GitHub 'Recent Deliveries' interface. At the top, there are three entries:

- A red warning icon next to a box containing the ID '4aeae280-4e38-11e4-9bac-c130e992644b' with the timestamp '2014-10-07 17:40:41'.
- A green checkmark icon next to a box containing the ID 'aff20880-4e37-11e4-9089-35319435e08b' with the timestamp '2014-10-07 17:36:21'.
- A green checkmark icon next to a box containing the ID '90f37680-4e37-11e4-9508-227d13b2ccfc' with the timestamp '2014-10-07 17:35:29'.

Below the list are tabs for 'Request' and 'Response'. The 'Response' tab is selected, showing a green button labeled '200' and a status message 'Completed in 0.61 seconds.' There is also a 'Redeliver' button.

The 'Headers' section displays the following request details:

```

Request URL: https://hooks.example.com/payload
Request method: POST
content-type: application/json
Expect:
User-Agent: GitHub-Hookshot/64a1910
X-Github-Delivery: 90f37680-4e37-11e4-9508-227d13b2ccfc
X-Github-Event: push
  
```

The 'Payload' section shows a JSON object representing a push event:

```

{
  "ref": "refs/heads/remove-whitespace",
  "before": "99d4fe5bffaf827f8a9e7cde00ccb0ab06a35e48",
  "after": "9370a6c3349331bac7e4c3c78c10bc8460c1e3e8",
  "created": false,
  "deleted": false,
  "forced": false,
  "base_ref": null,
  "compare": "https://github.com/tonychacon/fade/compare/99d4fe5bffaf...9370a6c33493",
  "commits": [
    {
      "id": "9370a6c3349331bac7e4c3c78c10bc8460c1e3e8",
      "distinct": true,
      "message": "remove whitespace",
      "timestamp": "2014-10-07T17:35:22+02:00",
      "url": "https://github.com/tonychacon/fade/commit/9370a6c3349331bac7e4c3c78c10bc8460"
    }
  ]
}
  
```

Otra cosa muy interesante es que puedes repetir el envío de cualquier petición para probar el servicio con facilidad.

Para más información sobre cómo escribir webhooks (enganches) y los diferentes tipos de eventos que puedes tratar, puedes ir a la documentación del desarrollador de GitHub, en: <https://developer.github.com/webhooks/>

La API de GitHub

Servicios y enganches nos sirven para recibir notificaciones “push” sobre eventos que suceden en tus repositorios. Pero, ¿qué pasa si necesitas más información acerca de estos eventos? ¿y si necesitas automatizar algo como añadir colaboradores o etiquetar incidencias?

Aquí es donde entra en juego la API de GitHub. GitHub tiene montones de llamadas de API para hacer casi cualquier cosa que puedes hacer vía web, de forma automatizada. En esta sección aprenderemos cómo autenticar y conectar a la API, cómo comentar una incidencia y cómo cambiar el estado de un Pull Request mediante la API.

Uso Básico

Lo más básico que podemos hacer es una petición GET a una llamada que no necesite autenticación. Por ejemplo, información solo lectura de un proyecto de código abierto. Por ejemplo, si queremos conocer información acerca del usuario “schacon”, podemos ejecutar algo como:

```
$ curl https://api.github.com/users/schacon
{
  "login": "schacon",
  "id": 70,
  "avatar_url": "https://avatars.githubusercontent.com/u/70",
  # ...
  "name": "Scott Chacon",
  "company": "GitHub",
  "following": 19,
  "created_at": "2008-01-27T17:19:28Z",
  "updated_at": "2014-06-10T02:37:23Z"
}
```

Hay muchísimas llamadas como esta para obtener información sobre organizaciones, proyectos, incidencias, commits, es decir, todo lo que podemos ver públicamente en la web de GitHub. Se puede usar la API para otras cosas como ver un fichero Markdown cualquier o encontrar una plantilla de `.gitignore`.

```
$ curl https://api.github.com/gitignore/templates/Java
{
  "name": "Java",
  "source": "*.class

# Mobile Tools for Java (J2ME)
.mtj.tmp/

# Package Files #
*.jar
*.war
*.ear

# virtual machine crash logs, see http://www.java.com/en/download/help/error_hotspot.xml
hs_err_pid*
```

"

}

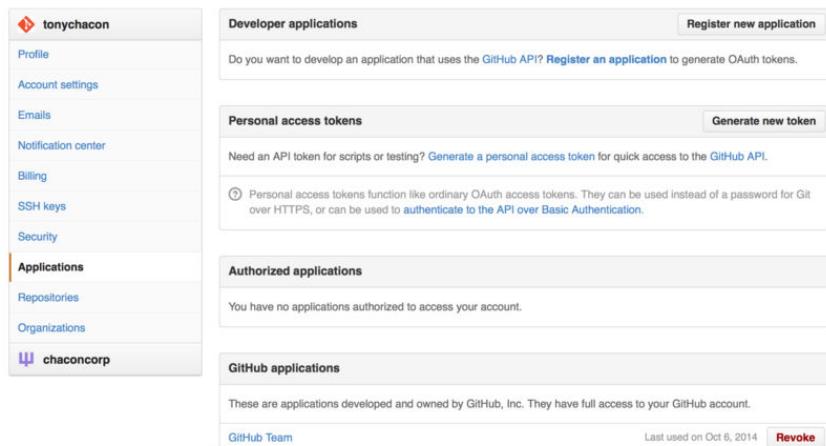
Comentarios en una incidencia

Sin embargo, si lo que quieres es realizar una acción como comentar una incidencia o un Pull Request, o si quieres ver o interactuar con un contenido privado, necesitas identificarte.

Hay varias formas de hacerlo. Puedes usar la autentificación básica, con tu usuario y tu contraseña, aunque generalmente es mejor usar un token de acceso personal. Puedes generararlo en la opción “Applications” de tu página de ajustes personales.

FIGURE 6-53

Generación del token de acceso.



Te preguntará acerca del ámbito que quieras para el token y una descripción. Asegúrate de usar una buena descripción para que te resulte fácil localizar aquellos token que ya no necesitas.

GitHub te permitirá ver el token una vez, por lo que tienes que copiarlo en ese momento. Ahora podrás identificarte en el script con el token, en lugar del usuario y la contraseña. Esto está bien porque puedes limitar el ámbito de lo que se quiere hacer y porque el token se puede anular.

También tiene la ventana de incrementar la tasa de accesos. Sin la autenticación podrás hacer 60 peticiones a la hora. Con una identificación el número de accesos permitidos sube a 5,000 por hora.

Realicemos entonces un comentario en una de nuestras incidencias. Por ejemplo, queremos dejar un comentario en la incidencia #6. Para ello, hacemos

una petición HTTP POST a `repos/<usuario>/<repo>/issues/<num>/comments` con el token que acabamos de generar como cabecera Authorization.

```
$ curl -H "Content-Type: application/json" \
-H "Authorization: token TOKEN" \
--data '{"body":"A new comment, :+1:"}' \
https://api.github.com/repos/schacon/blink/issues/6/comments
{
  "id": 58322100,
  "html_url": "https://github.com/schacon/blink/issues/6#issuecomment-58322100",
  ...
  "user": {
    "login": "tonychacon",
    "id": 7874698,
    "avatar_url": "https://avatars.githubusercontent.com/u/7874698?v=2",
    "type": "User",
  },
  "created_at": "2014-10-08T07:48:19Z",
  "updated_at": "2014-10-08T07:48:19Z",
  "body": "A new comment, :+1:"
}
```

Ahora, si vas a la incidencia, verás el comentario que acabas de enviar tal como en **Figure 6-54**.

**FIGURE 6-54**

Comentario enviado desde la API de GitHub.

Puedes usar la API para hacer todo lo que harías en el sitio web: crear y ajustar hitos, asignar gente a incidencias o pull requests, crear y cambiar etiquetas, acceder a datos de commit, crear nuevos commits y ramas, abrir, cerrar o fusionar Pull Requests, crear y editar equipos, comentar líneas de cambio en Pull Requests, buscar en el sitio y mucho más.

Cambio de estado de un Pull Request

Un ejemplo final que veremos es realmente útil si trabajas con Pull Requests. Cada commit tiene uno o más estados asociados con ellos, y hay una API para alterar y consultar ese estado.

Los servicios de integración continua y pruebas hacen uso de esta API para actuar cuando alguien envía código al repositorio, probando el mismo y devolviendo como resultado si el commit pasó todas las pruebas. Además, se podría comprobar si el mensaje del commit tiene un formato adecuado, si el autor siguió todas las recomendaciones para autores, si fue firmado, etc.

Supongamos que tenemos un enganche web en el repositorio que llama a un servicio web que comprueba si en el mensaje del commit aparece la cadena `Signed-off-by`.

```

require 'httpparty'
require 'sinatra'
require 'json'

post '/payload' do
  push = JSON.parse(request.body.read) # parse the JSON
  repo_name = push['repository']['full_name']

  # look through each commit message
  push["commits"].each do |commit|

    # look for a Signed-off-by string
    if /Signed-off-by/.match commit['message']
      state = 'success'
      description = 'Successfully signed off!'
    else
      state = 'failure'
      description = 'No signoff found.'
    end

    # post status to GitHub
    sha = commit["id"]
    status_url = "https://api.github.com/repos/#{repo_name}/statuses/#{sha}"

    status = {
      "state"      => state,
      "description" => description,
      "target_url"  => "http://example.com/how-to-signoff",
      "context"     => "validate/signoff"
    }
    HTTParty.post(status_url,
      :body => status.to_json,
      :headers => {
        'Content-Type'  => 'application/json',
        'User-Agent'    => 'tonychacon/signoff',
        'Authorization' => "token #{ENV['TOKEN']}" })
  end
end

```

Creemos que esto es fácil de seguir. En este controlador del enganche, miramos en cada commit enviado, y buscamos la cadena *Signed-off-by* en el mensaje de commit, y finalmente hacemos un HTTP POST al servicio de API /repos/<user>/<repo>/statuses/<commit_sha> con el resultado.

En este caso, puedes enviar un estado (*success*, *failure*, *error*), una descripción de qué ocurrió, un URL objetivo donde el usuario puede ir a buscar más información y un “contexto” en caso de que haya múltiples estados para un commit. Por ejemplo, un servicio de test puede dar un estado, y un servicio de validación puede dar por su parte su propio estado; el campo “context” serviría para diferenciarlos.

Si alguien abre un nuevo Pull Request en GitHub y este enganche está configurado, verías algo como **Figure 6-55**.

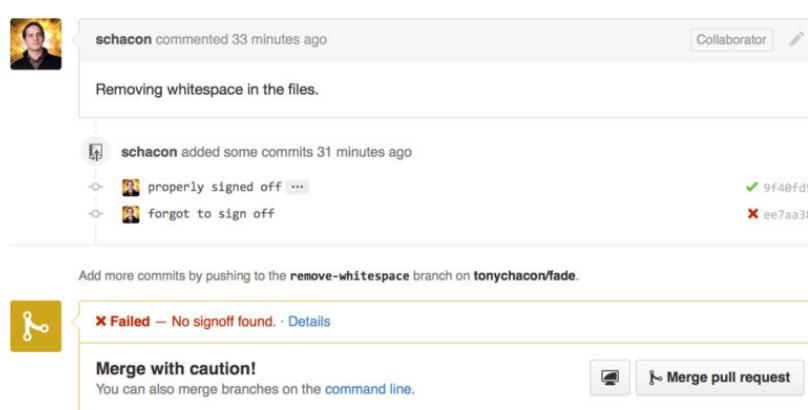


FIGURE 6-55

Estado del commit mediante API.

Podrás ver entonces una pequeña marca de color verde, que nos indica que el commit tiene la cadena “Signed-off-by” en el mensaje y un aspa roja en aquellos donde el autor olvidase hacer esa firma. También verías que el Pull Request toma el estado del último commit en la rama y te avisa de si es un fallo. Esto es realmente útil si usas la API para pruebas, y así evitar hacer una fusión accidental de unos commits que han fallado las pruebas.

Octokit

Hasta ahora hemos hecho casi todas las pruebas con `curl` y peticiones HTTP simples, pero en GitHub hay diferentes bibliotecas de código abierto que hacen más fácil el manejo de la API, agrupadas bajo el nombre de Octokit. En el momento de escribir esto, están soportados lenguajes como Go, Objective-C, Ruby

y .NET. Se puede ir a <http://github.com/octokit> para más información sobre esto, que te ayudarán a manejar peticiones y respuestas a la API de GitHub.

Con suerte estas utilidades te ayudarán a personalizar y modificar GitHub para trabajar mejor con tu forma concreta de trabajar. Para una documentación completa de la API así como ayudas para realizar tareas comunes, puedes consultar en <https://developer.github.com>.

Resumen

Ahora ya eres un usuario de GitHub. Ya sabes cómo crear una cuenta, gestionar una organización, crear y enviar repositorios, participar con los proyectos de otras personas y aceptar contribuciones de terceros en tus proyectos. En el siguiente capítulo conoceremos otras herramientas y trucos potentes para manejar situaciones más complicadas, con los que te puedes convertir con seguridad en un experto de Git.

Git Tools



By now, you've learned most of the day-to-day commands and workflows that you need to manage or maintain a Git repository for your source code control. You've accomplished the basic tasks of tracking and committing files, and you've harnessed the power of the staging area and lightweight topic branching and merging.

Now you'll explore a number of very powerful things that Git can do that you may not necessarily use on a day-to-day basis but that you may need at some point.

Revision Selection

Git allows you to specify specific commits or a range of commits in several ways. They aren't necessarily obvious but are helpful to know.

Single Revisions

You can obviously refer to a commit by the SHA-1 hash that it's given, but there are more human-friendly ways to refer to commits as well. This section outlines the various ways you can refer to a single commit.

Short SHA-1

Git is smart enough to figure out what commit you meant to type if you provide the first few characters, as long as your partial SHA-1 is at least four characters long and unambiguous – that is, only one object in the current repository begins with that partial SHA-1.

For example, to see a specific commit, suppose you run a `git log` command and identify the commit where you added certain functionality:

```
$ git log
commit 734713bc047d87bf7eac9674765ae793478c50d3
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Jan 2 18:32:33 2009 -0800

    fixed refs handling, added gc auto, updated tests

commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Merge: 1c002dd... 35cfb2b...
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 15:08:43 2008 -0800

    Merge commit 'phedders/rdocs'

commit 1c002dd4b536e7479fe34593e72e6c6c1819e53b
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 14:58:32 2008 -0800

    added some blame and merge stuff
```

In this case, choose `1c002dd....`. If you `git show` that commit, the following commands are equivalent (assuming the shorter versions are unambiguous):

```
$ git show 1c002dd4b536e7479fe34593e72e6c6c1819e53b
$ git show 1c002dd4b536e7479f
$ git show 1c002d
```

Git can figure out a short, unique abbreviation for your SHA-1 values. If you pass `--abbrev-commit` to the `git log` command, the output will use shorter values but keep them unique; it defaults to using seven characters but makes them longer if necessary to keep the SHA-1 unambiguous:

```
$ git log --abbrev-commit --pretty=oneline
ca82a6d changed the version number
085bb3b removed unnecessary test code
a11bef0 first commit
```

Generally, eight to ten characters are more than enough to be unique within a project.

As an example, the Linux kernel, which is a pretty large project with over 450k commits and 3.6 million objects, has no two objects whose SHA-1s overlap more than the first 11 characters.

A SHORT NOTE ABOUT SHA-1

A lot of people become concerned at some point that they will, by random happenstance, have two objects in their repository that hash to the same SHA-1 value. What then?

If you do happen to commit an object that hashes to the same SHA-1 value as a previous object in your repository, Git will see the previous object already in your Git database and assume it was already written. If you try to check out that object again at some point, you'll always get the data of the first object.

However, you should be aware of how ridiculously unlikely this scenario is. The SHA-1 digest is 20 bytes or 160 bits. The number of randomly hashed objects needed to ensure a 50% probability of a single collision is about 2^{80} (the formula for determining collision probability is $p = (n(n-1)/2) * (1/2^{160})$). 2^{80} is 1.2×10^{24} or 1 million billion billion. That's 1,200 times the number of grains of sand on the earth.

Here's an example to give you an idea of what it would take to get a SHA-1 collision. If all 6.5 billion humans on Earth were programming, and every second, each one was producing code that was the equivalent of the entire Linux kernel history (3.6 million Git objects) and pushing it into one enormous Git repository, it would take roughly 2 years until that repository contained enough objects to have a 50% probability of a single SHA-1 object collision. A higher probability exists that every member of your programming team will be attacked and killed by wolves in unrelated incidents on the same night.

Branch References

The most straightforward way to specify a commit requires that it has a branch reference pointed at it. Then, you can use a branch name in any Git command that expects a commit object or SHA-1 value. For instance, if you want to show the last commit object on a branch, the following commands are equivalent, assuming that the `topic1` branch points to `ca82a6d`:

```
$ git show ca82a6dff817ec66f44342007202690a93763949
$ git show topic1
```

If you want to see which specific SHA-1 a branch points to, or if you want to see what any of these examples boils down to in terms of SHA-1s, you can use a Git plumbing tool called `rev-parse`. You can see [Chapter 10](#) for more information about plumbing tools; basically, `rev-parse` exists for lower-level operations and isn't designed to be used in day-to-day operations. However, it can be

helpful sometimes when you need to see what's really going on. Here you can run `rev-parse` on your branch.

```
$ git rev-parse topic1
ca82a6dff817ec66f44342007202690a93763949
```

RefLog Shortnames

One of the things Git does in the background while you're working away is keep a “reflog” – a log of where your HEAD and branch references have been for the last few months.

You can see your reflog by using `git reflog`:

```
$ git reflog
734713b HEAD@{0}: commit: fixed refs handling, added gc auto, updated
d921970 HEAD@{1}: merge phedders/rdocs: Merge made by recursive.
1c002dd HEAD@{2}: commit: added some blame and merge stuff
1c36188 HEAD@{3}: rebase -i (squash): updating HEAD
95df984 HEAD@{4}: commit: # This is a combination of two commits.
1c36188 HEAD@{5}: rebase -i (squash): updating HEAD
7e05da5 HEAD@{6}: rebase -i (pick): updating HEAD
```

Every time your branch tip is updated for any reason, Git stores that information for you in this temporary history. And you can specify older commits with this data, as well. If you want to see the fifth prior value of the HEAD of your repository, you can use the `@{n}` reference that you see in the reflog output:

```
$ git show HEAD@{5}
```

You can also use this syntax to see where a branch was some specific amount of time ago. For instance, to see where your `master` branch was yesterday, you can type

```
$ git show master@{yesterday}
```

That shows you where the branch tip was yesterday. This technique only works for data that's still in your reflog, so you can't use it to look for commits older than a few months.

To see reflog information formatted like the `git log` output, you can run `git log -g`:

```
$ git log -g master
commit 734713bc047d87bf7eac9674765ae793478c50d3
Reflog: master@{0} (Scott Chacon <schacon@gmail.com>)
Reflog message: commit: fixed refs handling, added gc auto, updated
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Jan 2 18:32:33 2009 -0800

        fixed refs handling, added gc auto, updated tests

commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Reflog: master@{1} (Scott Chacon <schacon@gmail.com>)
Reflog message: merge phedders/rdocs: Merge made by recursive.
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 15:08:43 2008 -0800

Merge commit 'phedders/rdocs'
```

It's important to note that the reflog information is strictly local – it's a log of what you've done in your repository. The references won't be the same on someone else's copy of the repository; and right after you initially clone a repository, you'll have an empty reflog, as no activity has occurred yet in your repository. Running `git show HEAD@{2.months.ago}` will work only if you cloned the project at least two months ago – if you cloned it five minutes ago, you'll get no results.

Ancestry References

The other main way to specify a commit is via its ancestry. If you place a ^ at the end of a reference, Git resolves it to mean the parent of that commit. Suppose you look at the history of your project:

```
$ git log --pretty=format:'%h %s' --graph
* 734713b fixed refs handling, added gc auto, updated tests
*   d921970 Merge commit 'phedders/rdocs'
  \\
  | * 35cfb2b Some rdoc changes
  * | 1c002dd added some blame and merge stuff
  |
* 1c36188 ignore *.gem
* 9b29157 add open3_detach to gemspec file list
```

Then, you can see the previous commit by specifying HEAD[^], which means “the parent of HEAD”:

```
$ git show HEAD^
commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Merge: 1c002dd... 35cfb2b...
Author: Scott Chacon <schacon@gmail.com>
Date: Thu Dec 11 15:08:43 2008 -0800

Merge commit 'phedders/rdocs'
```

You can also specify a number after the ^ – for example, d921970^{^2} means “the second parent of d921970.” This syntax is only useful for merge commits, which have more than one parent. The first parent is the branch you were on when you merged, and the second is the commit on the branch that you merged in:

```
$ git show d921970^
commit 1c002dd4b536e7479fe34593e72e6c6c1819e53b
Author: Scott Chacon <schacon@gmail.com>
Date: Thu Dec 11 14:58:32 2008 -0800

added some blame and merge stuff

$ git show d921970^2
commit 35cfb2b795a55793d7cc56a6cc2060b4bb732548
Author: Paul Hedderly <paul+git@mjr.org>
Date: Wed Dec 10 22:22:03 2008 +0000

Some rdoc changes
```

The other main ancestry specification is the ~. This also refers to the first parent, so HEAD~ and HEAD[^] are equivalent. The difference becomes apparent when you specify a number. HEAD~2 means “the first parent of the first parent,” or “the grandparent” – it traverses the first parents the number of times you specify. For example, in the history listed earlier, HEAD~3 would be

```
$ git show HEAD~3
commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d
Author: Tom Preston-Werner <tom@mojombo.com>
Date: Fri Nov 7 13:47:59 2008 -0500

ignore *.gem
```

This can also be written HEAD^{^^^}, which again is the first parent of the first parent of the first parent:

```
$ git show HEAD^^^
commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d
Author: Tom Preston-Werner <tom@mojombo.com>
Date:   Fri Nov 7 13:47:59 2008 -0500

ignore *.gem
```

You can also combine these syntaxes – you can get the second parent of the previous reference (assuming it was a merge commit) by using HEAD~3^2, and so on.

Commit Ranges

Now that you can specify individual commits, let's see how to specify ranges of commits. This is particularly useful for managing your branches – if you have a lot of branches, you can use range specifications to answer questions such as, “What work is on this branch that I haven't yet merged into my main branch?”

DOUBLE DOT

The most common range specification is the double-dot syntax. This basically asks Git to resolve a range of commits that are reachable from one commit but aren't reachable from another. For example, say you have a commit history that looks like **Figure 7-1**.

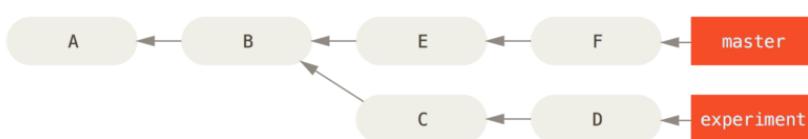


FIGURE 7-1

Example history for range selection.

You want to see what is in your experiment branch that hasn't yet been merged into your master branch. You can ask Git to show you a log of just those commits with `master..experiment` – that means “all commits reachable by experiment that aren't reachable by master.” For the sake of brevity and clarity

in these examples, I'll use the letters of the commit objects from the diagram in place of the actual log output in the order that they would display:

```
$ git log master..experiment
D
C
```

If, on the other hand, you want to see the opposite – all commits in `master` that aren't in `experiment` – you can reverse the branch names. `experiment..master` shows you everything in `master` not reachable from `experiment`:

```
$ git log experiment..master
F
E
```

This is useful if you want to keep the `experiment` branch up to date and preview what you're about to merge in. Another very frequent use of this syntax is to see what you're about to push to a remote:

```
$ git log origin/master..HEAD
```

This command shows you any commits in your current branch that aren't in the `master` branch on your `origin` remote. If you run a `git push` and your current branch is tracking `origin/master`, the commits listed by `git log origin/master..HEAD` are the commits that will be transferred to the server. You can also leave off one side of the syntax to have Git assume HEAD. For example, you can get the same results as in the previous example by typing `git log origin/master..` – Git substitutes HEAD if one side is missing.

MULTIPLE POINTS

The double-dot syntax is useful as a shorthand; but perhaps you want to specify more than two branches to indicate your revision, such as seeing what commits are in any of several branches that aren't in the branch you're currently on. Git allows you to do this by using either the `^` character or `--not` before any reference from which you don't want to see reachable commits. Thus these three commands are equivalent:

```
$ git log refA..refB
$ git log ^refA refB
$ git log refB --not refA
```

This is nice because with this syntax you can specify more than two references in your query, which you cannot do with the double-dot syntax. For instance, if you want to see all commits that are reachable from `refA` or `refB` but not from `refC`, you can type one of these:

```
$ git log refA refB ^refC
$ git log refA refB --not refC
```

This makes for a very powerful revision query system that should help you figure out what is in your branches.

TRIPLE DOT

The last major range-selection syntax is the triple-dot syntax, which specifies all the commits that are reachable by either of two references but not by both of them. Look back at the example commit history in [Figure 7-1](#). If you want to see what is in `master` or `experiment` but not any common references, you can run

```
$ git log master...experiment
F
E
D
C
```

Again, this gives you normal `log` output but shows you only the commit information for those four commits, appearing in the traditional commit date ordering.

A common switch to use with the `log` command in this case is `--left-right`, which shows you which side of the range each commit is in. This helps make the data more useful:

```
$ git log --left-right master...experiment
< F
< E
```

```
> D
> C
```

With these tools, you can much more easily let Git know what commit or commits you want to inspect.

Interactive Staging

Git comes with a couple of scripts that make some command-line tasks easier. Here, you'll look at a few interactive commands that can help you easily craft your commits to include only certain combinations and parts of files. These tools are very helpful if you modify a bunch of files and then decide that you want those changes to be in several focused commits rather than one big messy commit. This way, you can make sure your commits are logically separate changesets and can be easily reviewed by the developers working with you. If you run `git add` with the `-i` or `--interactive` option, Git goes into an interactive shell mode, displaying something like this:

```
$ git add -i
      staged      unstaged path
 1: unchanged      +0/-1 TODO
 2: unchanged      +1/-1 index.html
 3: unchanged      +5/-1 lib/simplegit.rb

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

You can see that this command shows you a much different view of your staging area – basically the same information you get with `git status` but a bit more succinct and informative. It lists the changes you've staged on the left and unstaged changes on the right.

After this comes a Commands section. Here you can do a number of things, including staging files, unstaging files, staging parts of files, adding untracked files, and seeing diffs of what has been staged.

Staging and Unstaging Files

If you type 2 or u at the `What now>` prompt, the script prompts you for which files you want to stage:

```
What now> 2
      staged      unstaged path
1:   unchanged      +0/-1 TODO
2:   unchanged      +1/-1 index.html
3:   unchanged      +5/-1 lib/simplegit.rb
Update>>
```

To stage the TODO and index.html files, you can type the numbers:

```
Update>> 1,2
      staged      unstaged path
* 1:   unchanged      +0/-1 TODO
* 2:   unchanged      +1/-1 index.html
3:   unchanged      +5/-1 lib/simplegit.rb
Update>>
```

The * next to each file means the file is selected to be staged. If you press Enter after typing nothing at the Update>> prompt, Git takes anything selected and stages it for you:

```
Update>>
updated 2 paths

*** Commands ***
1: status      2: update      3: revert      4: add untracked
5: patch      6: diff        7: quit        8: help
What now> 1
      staged      unstaged path
1:      +0/-1      nothing TODO
2:      +1/-1      nothing index.html
3:   unchanged      +5/-1 lib/simplegit.rb
```

Now you can see that the TODO and index.html files are staged and the simplegit.rb file is still unstaged. If you want to unstage the TODO file at this point, you use the 3 or r (for revert) option:

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

```

3:    unchanged      +5/-1 lib/simplegit.rb
Revert>> 1
      staged      unstaged path
* 1:    +0/-1      nothing TODO
  2:    +1/-1      nothing index.html
  3:    unchanged      +5/-1 lib/simplegit.rb
Revert>> [enter]
reverted one path

```

Looking at your Git status again, you can see that you've unstaged the TODO file:

```

*** Commands ***
1: status      2: update      3: revert      4: add untracked
5: patch      6: diff        7: quit        8: help
What now> 1
      staged      unstaged path
1:    unchanged      +0/-1 TODO
  2:    +1/-1      nothing index.html
  3:    unchanged      +5/-1 lib/simplegit.rb

```

To see the diff of what you've staged, you can use the 6 or d (for diff) command. It shows you a list of your staged files, and you can select the ones for which you would like to see the staged diff. This is much like specifying `git diff --cached` on the command line:

```

*** Commands ***
1: status      2: update      3: revert      4: add untracked
5: patch      6: diff        7: quit        8: help
What now> 6
      staged      unstaged path
1:    +1/-1      nothing index.html
Review diff>> 1
diff --git a/index.html b/index.html
index 4d07108..4335f49 100644
--- a/index.html
+++ b/index.html
@@ -16,7 +16,7 @@ Date Finder

<p id="out">...</p>

-<div id="footer">contact : support@github.com</div>
+<div id="footer">contact : email.support@github.com</div>

<script type="text/javascript">

```

With these basic commands, you can use the interactive add mode to deal with your staging area a little more easily.

Staging Patches

It's also possible for Git to stage certain parts of files and not the rest. For example, if you make two changes to your simplegit.rb file and want to stage one of them and not the other, doing so is very easy in Git. From the interactive prompt, type `5` or `p` (for patch). Git will ask you which files you would like to partially stage; then, for each section of the selected files, it will display hunks of the file diff and ask if you would like to stage them, one by one:

```
diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index dd5ecc4..57399e0 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -22,7 +22,7 @@ class SimpleGit
  end

  def log(treeish = 'master')
-    command("git log -n 25 #{treeish}")
+    command("git log -n 30 #{treeish}")
  end

  def blame(path)
Stage this hunk [y,n,a,d,/ ,j,J,g,e,?]?
```

You have a lot of options at this point. Typing `?` shows a list of what you can do:

```
Stage this hunk [y,n,a,d,/ ,j,J,g,e,?]? ?
y - stage this hunk
n - do not stage this hunk
a - stage this and all the remaining hunks in the file
d - do not stage this hunk nor any of the remaining hunks in the file
g - select a hunk to go to
/ - search for a hunk matching the given regex
j - leave this hunk undecided, see next undecided hunk
J - leave this hunk undecided, see next hunk
k - leave this hunk undecided, see previous undecided hunk
K - leave this hunk undecided, see previous hunk
s - split the current hunk into smaller hunks
e - manually edit the current hunk
? - print help
```

Generally, you'll type `y` or `n` if you want to stage each hunk, but staging all of them in certain files or skipping a hunk decision until later can be helpful too. If you stage one part of the file and leave another part unstaged, your status output will look like this:

```
What now> 1
      staged     unstaged path
1:   unchanged      +0/-1 TODO
2:       +1/-1    nothing index.html
3:       +1/-1      +4/-0 lib/simplegit.rb
```

The status of the `simplegit.rb` file is interesting. It shows you that a couple of lines are staged and a couple are unstaged. You've partially staged this file. At this point, you can exit the interactive adding script and run `git commit` to commit the partially staged files.

You also don't need to be in interactive add mode to do the partial-file staging – you can start the same script by using `git add -p` or `git add --patch` on the command line.

Furthermore, you can use patch mode for partially resetting files with the `reset --patch` command, for checking out parts of files with the `checkout --patch` command and for stashing parts of files with the `stash save --patch` command. We'll go into more details on each of these as we get to more advanced usages of these commands.

Stashing and Cleaning

Often, when you've been working on part of your project, things are in a messy state and you want to switch branches for a bit to work on something else. The problem is, you don't want to do a commit of half-done work just so you can get back to this point later. The answer to this issue is the `git stash` command.

Stashing takes the dirty state of your working directory – that is, your modified tracked files and staged changes – and saves it on a stack of unfinished changes that you can reapply at any time.

Stashing Your Work

To demonstrate, you'll go into your project and start working on a couple of files and possibly stage one of the changes. If you run `git status`, you can see your dirty state:

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

    modified:   index.html

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

    modified:   lib/simplegit.rb
```

Now you want to switch branches, but you don't want to commit what you've been working on yet; so you'll stash the changes. To push a new stash onto your stack, run `git stash` or `git stash save`:

```
$ git stash
Saved working directory and index state \
"WIP on master: 049d078 added the index file"
HEAD is now at 049d078 added the index file
(To restore them type "git stash apply")
```

Your working directory is clean:

```
$ git status
# On branch master
nothing to commit, working directory clean
```

At this point, you can easily switch branches and do work elsewhere; your changes are stored on your stack. To see which stashes you've stored, you can use `git stash list`:

```
$ git stash list
stash@{0}: WIP on master: 049d078 added the index file
stash@{1}: WIP on master: c264051 Revert "added file_size"
stash@{2}: WIP on master: 21d80a5 added number to log
```

In this case, two stashes were done previously, so you have access to three different stashed works. You can reapply the one you just stashed by using the command shown in the help output of the original stash command: `git stash apply`. If you want to apply one of the older stashes, you can specify it by nam-

ing it, like this: `git stash apply stash@{2}`. If you don't specify a stash, Git assumes the most recent stash and tries to apply it:

```
$ git stash apply
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#       modified:   index.html
#       modified:   lib/simplegit.rb
#
```

You can see that Git re-modifies the files you reverted when you saved the stash. In this case, you had a clean working directory when you tried to apply the stash, and you tried to apply it on the same branch you saved it from; but having a clean working directory and applying it on the same branch aren't necessary to successfully apply a stash. You can save a stash on one branch, switch to another branch later, and try to reapply the changes. You can also have modified and uncommitted files in your working directory when you apply a stash – Git gives you merge conflicts if anything no longer applies cleanly.

The changes to your files were reapplied, but the file you staged before wasn't restaged. To do that, you must run the `git stash apply` command with a `--index` option to tell the command to try to reapply the staged changes. If you had run that instead, you'd have gotten back to your original position:

```
$ git stash apply --index
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   index.html
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#       modified:   lib/simplegit.rb
#
```

The `apply` option only tries to apply the stashed work – you continue to have it on your stack. To remove it, you can run `git stash drop` with the name of the stash to remove:

```
$ git stash list
stash@{0}: WIP on master: 049d078 added the index file
stash@{1}: WIP on master: c264051 Revert "added file_size"
stash@{2}: WIP on master: 21d80a5 added number to log
$ git stash drop stash@{0}
Dropped stash@{0} (364e91f3f268f0900bc3ee613f9f733e82aaed43)
```

You can also run `git stash pop` to apply the stash and then immediately drop it from your stack.

Creative Stashing

There are a few stash variants that may also be helpful. The first option that is quite popular is the `--keep-index` option to the `stash save` command. This tells Git to not stash anything that you've already staged with the `git add` command.

This can be really helpful if you've made a number of changes but want to only commit some of them and then come back to the rest of the changes at a later time.

```
$ git status -s
M index.html
M lib/simplegit.rb

$ git stash --keep-index
Saved working directory and index state WIP on master: 1b65b17 added the index file
HEAD is now at 1b65b17 added the index file

$ git status -s
M index.html
```

Another common thing you may want to do with stash is to stash the untracked files as well as the tracked ones. By default, `git stash` will only store files that are already in the index. If you specify `--include-untracked` or `-u`, Git will also stash any untracked files you have created.

```
$ git status -s
M index.html
M lib/simplegit.rb
?? new-file.txt

$ git stash -u
```

```
 Saved working directory and index state WIP on master: 1b65b17 added the index file
HEAD is now at 1b65b17 added the index file
```

```
$ git status -s
$
```

Finally, if you specify the `--patch` flag, Git will not stash everything that is modified but will instead prompt you interactively which of the changes you would like to stash and which you would like to keep in your working directly.

```
$ git stash --patch
diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index 66d332e..8bb5674 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -16,6 +16,10 @@ class SimpleGit
      return `#{git_cmd} 2>&1`.chomp
    end
  end
+
+ def show(treeish = 'master')
+   command("git show #{treeish}")
+ end

end
test
Stash this hunk [y,n,q,a,d,/ ,e,?]?
```

```
 Saved working directory and index state WIP on master: 1b65b17 added the index file
```

Creating a Branch from a Stash

If you stash some work, leave it there for a while, and continue on the branch from which you stashed the work, you may have a problem reapplying the work. If the apply tries to modify a file that you've since modified, you'll get a merge conflict and will have to try to resolve it. If you want an easier way to test the stashed changes again, you can run `git stash branch`, which creates a new branch for you, checks out the commit you were on when you stashed your work, reapplies your work there, and then drops the stash if it applies successfully:

```
$ git stash branch testchanges
Switched to a new branch "testchanges"
# On branch testchanges
```

```
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   index.html
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#       modified:   lib/simplegit.rb
#
Dropped refs/stash@{0} (f0dfc4d5dc332d1cee34a634182e168c4efc3359)
```

This is a nice shortcut to recover stashed work easily and work on it in a new branch.

Cleaning your Working Directory

Finally, you may not want to stash some work or files in your working directory, but simply get rid of them. The `git clean` command will do this for you.

Some common reasons for this might be to remove cruft that has been generated by merges or external tools or to remove build artifacts in order to run a clean build.

You'll want to be pretty careful with this command, since it's designed to remove files from your working directory that are not tracked. If you change your mind, there is often no retrieving the content of those files. A safer option is to run `git stash --all` to remove everything but save it in a stash.

Assuming you do want to remove cruft files or clean your working directory, you can do so with `git clean`. To remove all the untracked files in your working directory, you can run `git clean -f -d`, which removes any files and also any subdirectories that become empty as a result. The `-f` means *force* or “really do this”.

If you ever want to see what it would do, you can run the command with the `-n` option, which means “do a dry run and tell me what you *would* have removed”.

```
$ git clean -d -n
Would remove test.o
Would remove tmp/
```

By default, the `git clean` command will only remove untracked files that are not ignored. Any file that matches a pattern in your `.gitignore` or other ignore files will not be removed. If you want to remove those files too, such as

to remove all .o files generated from a build so you can do a fully clean build, you can add a -x to the clean command.

```
$ git status -s
M lib/simplegit.rb
?? build.TMP
?? tmp/

$ git clean -n -d
Would remove build.TMP
Would remove tmp/

$ git clean -n -d -x
Would remove build.TMP
Would remove test.o
Would remove tmp/
```

If you don't know what the `git clean` command is going to do, always run it with a `-n` first to double check before changing the `-n` to a `-f` and doing it for real. The other way you can be careful about the process is to run it with the `-i` or "interactive" flag.

This will run the clean command in an interactive mode.

```
$ git clean -x -i
Would remove the following items:
  build.TMP  test.o
*** Commands ***
  1: clean           2: filter by pattern   3: select by numbers  4: ask
  6: help
What now>
```

This way you can step through each file individually or specify patterns for deletion interactively.

Signing Your Work

Git is cryptographically secure, but it's not foolproof. If you're taking work from others on the internet and want to verify that commits are actually from a trusted source, Git has a few ways to sign and verify work using GPG.

GPG Introduction

First of all, if you want to sign anything you need to get GPG configured and your personal key installed.

```
$ gpg --list-keys  
/Users/schacon/.gnupg/pubring.gpg  
-----  
pub 2048R/0A46826A 2014-06-04  
uid Scott Chacon (Git signing key) <schacon@gmail.com>  
sub 2048R/874529A9 2014-06-04
```

If you don't have a key installed, you can generate one with `gpg --gen-key`.

```
gpg --gen-key
```

Once you have a private key to sign with, you can configure Git to use it for signing things by setting the `user.signingkey` config setting.

```
git config --global user.signingkey 0A46826A
```

Now Git will use your key by default to sign tags and commits if you want.

Signing Tags

If you have a GPG private key setup, you can now use it to sign new tags. All you have to do is use `-s` instead of `-a`:

```
$ git tag -s v1.5 -m 'my signed 1.5 tag'
```

```
You need a passphrase to unlock the secret key for
user: "Ben Straub <ben@straub.cc>"
2048-bit RSA key, ID 800430EB, created 2014-05-04
```

If you run `git show` on that tag, you can see your GPG signature attached to it:

```
$ git show v1.5
tag v1.5
Tagger: Ben Straub <ben@straub.cc>
Date:   Sat May 3 20:29:41 2014 -0700

my signed 1.5 tag
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1

iQEcBAABAgAGBQJTZbQlAAoJEF0+sviABDDrZbQH/09PFE51KPVPlanr6q1v4/Ut
LQxfojUWiLQdg2ESJItkcuweYg+kC3HCyFejeDIBw9dpXt00rY26p05qrpnG+85b
hM1/PswpPLuBSr+oCIDj5GMC2r2iEKsfv2fJbNW8iWAXVLoWZRF8B0MFqX/YTMbm
ecorc4ixzQu7tupRihslbNkfvcimnSDeSvzCpWAhl7h8Wj6hhqePmLm9lAYqnKp
8S5B/1SSQuEAjRZgI4IexpZoeKGDptPHxLLS38fozsyi0QyDyzEgJxcJQVMXxVi
RUysgqjcpT8+iQM1PblGfHR4XAhUoqN5Fx06PSaFZhqvWFezJ28/CLyX5q+oIVk=
=EFTF
-----END PGP SIGNATURE-----

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

    changed the version number
```

Verifying Tags

To verify a signed tag, you use `git tag -v [tag-name]`. This command uses GPG to verify the signature. You need the signer's public key in your keyring for this to work properly:

```
$ git tag -v v1.4.2.1
object 883653babd8ee7ea23e6a5c392bb739348b1eb61
type commit
tag v1.4.2.1
tagger Junio C Hamano <junkio@cox.net> 1158138501 -0700

GIT 1.4.2.1

Minor fixes since 1.4.2, including git-mv and git-http with alternates.
gpg: Signature made Wed Sep 13 02:08:25 2006 PDT using DSA key ID F3119B9A
gpg: Good signature from "Junio C Hamano <junkio@cox.net>"
```

gpg: aka "[jpeg image of size 1513]"

Primary key fingerprint: 3565 2A26 2040 E066 C9A7 4A7D C0C6 D9A4 F311 9B9A

If you don't have the signer's public key, you get something like this instead:

```
gpg: Signature made Wed Sep 13 02:08:25 2006 PDT using DSA key ID F3119B9A
gpg: Can't check signature: public key not found
error: could not verify the tag 'v1.4.2.1'
```

Signing Commits

In more recent versions of Git (v1.7.9 and above), you can now also sign individual commits. If you're interested in signing commits directly instead of just the tags, all you need to do is add a `-S` to your `git commit` command.

```
$ git commit -a -S -m 'signed commit'

You need a passphrase to unlock the secret key for
user: "Scott Chacon (Git signing key) <schacon@gmail.com>"
2048-bit RSA key, ID 0A46826A, created 2014-06-04

[master 5c3386c] signed commit
 4 files changed, 4 insertions(+), 24 deletions(-)
 rewrite Rakefile (100%)
 create mode 100644 lib/git.rb
```

To see and verify these signatures, there is also a `--show-signature` option to `git log`.

```
$ git log --show-signature -1
commit 5c3386cf54bba0a33a32da706aa52bc0155503c2
gpg: Signature made Wed Jun 4 19:49:17 2014 PDT using RSA key ID 0A46826A
gpg: Good signature from "Scott Chacon (Git signing key) <schacon@gmail.com>"
```

Author: Scott Chacon <schacon@gmail.com>
Date: Wed Jun 4 19:49:17 2014 -0700

```
        signed commit
```

Additionally, you can configure `git log` to check any signatures it finds and list them in its output with the `%G?` format.

```
$ git log --pretty=format:"%h %G? %aN %s"
5c3386c G Scott Chacon signed commit
ca82a6d N Scott Chacon changed the version number
```

```
085bb3b N Scott Chacon removed unnecessary test code
a11bef0 N Scott Chacon first commit
```

Here we can see that only the latest commit is signed and valid and the previous commits are not.

In Git 1.8.3 and later, “git merge” and “git pull” can be told to inspect and reject when merging a commit that does not carry a trusted GPG signature with the `--verify-signatures` command.

If you use this option when merging a branch and it contains commits that are not signed and valid, the merge will not work.

```
$ git merge --verify-signatures non-verify
fatal: Commit ab06180 does not have a GPG signature.
```

If the merge contains only valid signed commits, the merge command will show you all the signatures it has checked and then move forward with the merge.

```
$ git merge --verify-signatures signed-branch
Commit 13ad65e has a good GPG signature by Scott Chacon (Git signing key) <schacon@gmail.com>
Updating 5c3386c..13ad65e
Fast-forward
 README | 2 ++
 1 file changed, 2 insertions(+)
```

You can also use the `-S` option with the `git merge` command itself to sign the resulting merge commit itself. The following example both verifies that every commit in the branch to be merged is signed and furthermore signs the resulting merge commit.

```
$ git merge --verify-signatures -S signed-branch
Commit 13ad65e has a good GPG signature by Scott Chacon (Git signing key) <schacon@gmail.com>

You need a passphrase to unlock the secret key for
user: "Scott Chacon (Git signing key) <schacon@gmail.com>"
2048-bit RSA key, ID 0A46826A, created 2014-06-04

Merge made by the 'recursive' strategy.
 README | 2 ++
 1 file changed, 2 insertions(+)
```

Everyone Must Sign

Sigining tags and commits is great, but if you decide to use this in your normal workflow, you'll have to make sure that everyone on your team understands how to do so. If you don't, you'll end up spending a lot of time helping people figure out how to rewrite their commits with signed versions. Make sure you understand GPG and the benefits of signing things before adopting this as part of your standard workflow.

Searching

With just about any size codebase, you'll often need to find where a function is called or defined, or find the history of a method. Git provides a couple of useful tools for looking through the code and commits stored in it's database quickly and easily. We'll go through a few of them.

Git Grep

Git ships with a command called `grep` that allows you to easily search through any committed tree or the working directory for a string or regular expression. For these examples, we'll look through the Git source code itself.

By default, it will look through the files in your working directory. You can pass `-n` to print out the line numbers where Git has found matches.

```
$ git grep -n gmtime_r
compat/gmtime.c:3:#undef gmtime_r
compat/gmtime.c:8:        return git_gmtime_r(timep, &result);
compat/gmtime.c:11:struct tm *git_gmtime_r(const time_t *timep, struct tm *result)
compat/gmtime.c:16:    ret = gmtime_r(timep, result);
compat/mingw.c:606:struct tm *gmtime_r(const time_t *timep, struct tm *result)
compat/mingw.h:162:struct tm *gmtime_r(const time_t *timep, struct tm *result);
date.c:429:            if (gmtime_r(&now, &now_tm))
date.c:492:            if (gmtime_r(&time, tm)) {
git-compat-util.h:721:struct tm *git_gmtime_r(const time_t *, struct tm *);
git-compat-util.h:723:#define gmtime_r git_gmtime_r
```

There are a number of interesting options you can provide the `grep` command.

For instance, instead of the previous call, you can have Git summarize the output by just showing you which files matched and how many matches there were in each file with the `--count` option:

```
$ git grep --count gmtime_r
compat/gmtime.c:4
compat/mingw.c:1
compat/mingw.h:1
date.c:2
git-compat-util.h:2
```

If you want to see what method or function it thinks it has found a match in, you can pass `-p`:

```
$ git grep -p gmtime_r *.c
date.c:static int match_multi_number(unsigned long num, char c, const char *date,
date.c:           if (gmtime_r(&now, &now_tm))
date.c:static int match_digit(const char *date, struct tm *tm, int *offset, int *t
date.c:           if (gmtime_r(&time, tm)) {
```

So here we can see that `gmtime_r` is called in the `match_multi_number` and `match_digit` functions in the `date.c` file.

You can also look for complex combinations of strings with the `--and` flag, which makes sure that multiple matches are in the same line. For instance, let's look for any lines that define a constant with either the strings "LINK" or "BUF_MAX" in them in the Git codebase in an older 1.8.0 version.

Here we'll also use the `--break` and `--heading` options which help split up the output into a more readable format.

```
$ git grep --break --heading \
    -n -e '#define' --and \(
        -e LINK -e BUF_MAX \
    \) v1.8.0
v1.8.0:builtin/index-pack.c
62:#define FLAG_LINK (1u<<20)

v1.8.0:cache.h
73:#define S_IFGITLINK 0160000
74:#define S_ISGITLINK(m) (((m) & S_IFMT) == S_IFGITLINK)

v1.8.0:environment.c
54:#define OBJECT_CREATION_MODE OBJECT_CREATIONUSES_HARDLINKS

v1.8.0:strbuf.c
326:#define STRBUF_MAXLINK (2*PATH_MAX)

v1.8.0:symlinks.c
53:#define FL_SYMLINK (1 << 2)

v1.8.0:zlib.c
```

```
30:/* #define ZLIB_BUF_MAX ((uInt)-1) */
31:#define ZLIB_BUF_MAX ((uInt) 1024 * 1024 * 1024) /* 1GB */
```

The `git grep` command has a few advantages over normal searching commands like `grep` and `ack`. The first is that it's really fast, the second is that you can search through any tree in Git, not just the working directory. As we saw in the above example, we looked for terms in an older version of the Git source code, not the version that was currently checked out.

Git Log Searching

Perhaps you're looking not for **where** a term exists, but **when** it existed or was introduced. The `git log` command has a number of powerful tools for finding specific commits by the content of their messages or even the content of the diff they introduce.

If we want to find out for example when the `ZLIB_BUF_MAX` constant was originally introduced, we can tell Git to only show us the commits that either added or removed that string with the `-S` option.

```
$ git log -S ZLIB_BUF_MAX --oneline
e01503b zlib: allow feeding more than 4GB in one go
ef49a7a zlib: zlib can only process 4GB at a time
```

If we look at the diff of those commits we can see that in `ef49a7a` the constant was introduced and in `e01503b` it was modified.

If you need to be more specific, you can provide a regular expression to search for with the `-G` option.

LINE LOG SEARCH

Another fairly advanced log search that is insanely useful is the line history search. This is a fairly recent addition and not very well known, but it can be really helpful. It is called with the `-L` option to `git log` and will show you the history of a function or line of code in your codebase.

For example, if we wanted to see every change made to the function `git_deflate_bound` in the `zlib.c` file, we could run `git log -L :git_deflate_bound:zlib.c`. This will try to figure out what the bounds of that function are and then look through the history and show us every change that was made to the function as a series of patches back to when the function was first created.

```
$ git log -L :git_deflate_bound:zlib.c
commit ef49a7a0126d64359c974b4b3b71d7ad42ee3bca
Author: Junio C Hamano <gitster@pobox.com>
Date:   Fri Jun 10 11:52:15 2011 -0700

    zlib: zlib can only process 4GB at a time

diff --git a/zlib.c b/zlib.c
--- a/zlib.c
+++ b/zlib.c
@@ -85,5 +130,5 @@
-unsigned long git_deflate_bound(z_streamp strm, unsigned long size)
+unsigned long git_deflate_bound(git_zstream *strm, unsigned long size)
{
-    return deflateBound(strm, size);
+    return deflateBound(&strm->z, size);
}

commit 225a6f1068f71723a910e8565db4e252b3ca21fa
Author: Junio C Hamano <gitster@pobox.com>
Date:   Fri Jun 10 11:18:17 2011 -0700

    zlib: wrap deflateBound() too

diff --git a/zlib.c b/zlib.c
--- a/zlib.c
+++ b/zlib.c
@@ -81,0 +85,5 @@
+unsigned long git_deflate_bound(z_streamp strm, unsigned long size)
+{
+    return deflateBound(strm, size);
+}
+
```

If Git can't figure out how to match a function or method in your programming language, you can also provide it a regex. For example, this would have done the same thing: `git log -L '/unsigned long git_deflate_bound/' ,/^}:/zlib.c`. You could also give it a range of lines or a single line number and you'll get the same sort of output.

Rewriting History

Many times, when working with Git, you may want to revise your commit history for some reason. One of the great things about Git is that it allows you to

make decisions at the last possible moment. You can decide what files go into which commits right before you commit with the staging area, you can decide that you didn't mean to be working on something yet with the stash command, and you can rewrite commits that already happened so they look like they happened in a different way. This can involve changing the order of the commits, changing messages or modifying files in a commit, squashing together or splitting apart commits, or removing commits entirely – all before you share your work with others.

In this section, you'll cover how to accomplish these very useful tasks so that you can make your commit history look the way you want before you share it with others.

Changing the Last Commit

Changing your last commit is probably the most common rewriting of history that you'll do. You'll often want to do two basic things to your last commit: change the commit message, or change the snapshot you just recorded by adding, changing and removing files.

If you only want to modify your last commit message, it's very simple:

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

That drops you into your text editor, which has your last commit message in it, ready for you to modify the message. When you save and close the editor, the editor writes a new commit containing that message and makes it your new last commit.

If you've committed and then you want to change the snapshot you committed by adding or changing files, possibly because you forgot to add a newly created file when you originally committed, the process works basically the same way. You stage the changes you want by editing a file and running `git add` on it or `git rm` to a tracked file, and the subsequent `git commit --amend` takes your current staging area and makes it the snapshot for the new commit.

You need to be careful with this technique because amending changes the SHA-1 of the commit. It's like a very small rebase – don't amend your last commit if you've already pushed it.

Changing Multiple Commit Messages

To modify a commit that is farther back in your history, you must move to more complex tools. Git doesn't have a modify-history tool, but you can use the re-

base tool to rebase a series of commits onto the HEAD they were originally based on instead of moving them to another one. With the interactive rebase tool, you can then stop after each commit you want to modify and change the message, add files, or do whatever you wish. You can run rebase interactively by adding the `-i` option to `git rebase`. You must indicate how far back you want to rewrite commits by telling the command which commit to rebase onto.

For example, if you want to change the last three commit messages, or any of the commit messages in that group, you supply as an argument to `git rebase -i` the parent of the last commit you want to edit, which is `HEAD~2^` or `HEAD~3`. It may be easier to remember the `~3` because you're trying to edit the last three commits; but keep in mind that you're actually designating four commits ago, the parent of the last commit you want to edit:

```
$ git rebase -i HEAD~3
```

Remember again that this is a rebasing command – every commit included in the range `HEAD~3..HEAD` will be rewritten, whether you change the message or not. Don't include any commit you've already pushed to a central server – doing so will confuse other developers by providing an alternate version of the same change.

Running this command gives you a list of commits in your text editor that looks something like this:

```
pick f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file

# Rebase 710f0f8..a5f4a0d onto 710f0f8
#
# Commands:
#   p, pick = use commit
#   r, reword = use commit, but edit the commit message
#   e, edit = use commit, but stop for amending
#   s, squash = use commit, but meld into previous commit
#   f, fixup = like "squash", but discard this commit's log message
#   x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
```

```
#  
# Note that empty commits are commented out
```

It's important to note that these commits are listed in the opposite order than you normally see them using the `log` command. If you run a `log`, you see something like this:

```
$ git log --pretty=format:"%h %s" HEAD~3..HEAD  
a5f4a0d added cat-file  
310154e updated README formatting and added blame  
f7f3f6d changed my name a bit
```

Notice the reverse order. The interactive rebase gives you a script that it's going to run. It will start at the commit you specify on the command line (`HEAD~3`) and replay the changes introduced in each of these commits from top to bottom. It lists the oldest at the top, rather than the newest, because that's the first one it will replay.

You need to edit the script so that it stops at the commit you want to edit. To do so, change the word 'pick' to the word 'edit' for each of the commits you want the script to stop after. For example, to modify only the third commit message, you change the file to look like this:

```
edit f7f3f6d changed my name a bit  
pick 310154e updated README formatting and added blame  
pick a5f4a0d added cat-file
```

When you save and exit the editor, Git rewinds you back to the last commit in that list and drops you on the command line with the following message:

```
$ git rebase -i HEAD~3  
Stopped at f7f3f6d... changed my name a bit  
You can amend the commit now, with
```

```
git commit --amend
```

Once you're satisfied with your changes, run

```
git rebase --continue
```

These instructions tell you exactly what to do. Type

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

Change the commit message, and exit the editor. Then, run

```
$ git rebase --continue
```

This command will apply the other two commits automatically, and then you’re done. If you change pick to edit on more lines, you can repeat these steps for each commit you change to edit. Each time, Git will stop, let you amend the commit, and continue when you’re finished.

Reordering Commits

You can also use interactive rebases to reorder or remove commits entirely. If you want to remove the “added cat-file” commit and change the order in which the other two commits are introduced, you can change the rebase script from this

```
pick f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

to this:

```
pick 310154e updated README formatting and added blame
pick f7f3f6d changed my name a bit
```

When you save and exit the editor, Git rewinds your branch to the parent of these commits, applies 310154e and then f7f3f6d, and then stops. You effectively change the order of those commits and remove the “added cat-file” commit completely.

Squashing Commits

It’s also possible to take a series of commits and squash them down into a single commit with the interactive rebasing tool. The script puts helpful instructions in the rebase message:

```

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

```

If, instead of “pick” or “edit”, you specify “squash”, Git applies both that change and the change directly before it and makes you merge the commit messages together. So, if you want to make a single commit from these three commits, you make the script look like this:

```

pick f7f3f6d changed my name a bit
squash 310154e updated README formatting and added blame
squash a5f4a0d added cat-file

```

When you save and exit the editor, Git applies all three changes and then puts you back into the editor to merge the three commit messages:

```

# This is a combination of 3 commits.
# The first commit's message is:
changed my name a bit

# This is the 2nd commit message:

updated README formatting and added blame

# This is the 3rd commit message:

added cat-file

```

When you save that, you have a single commit that introduces the changes of all three previous commits.

Splitting a Commit

Splitting a commit undoes a commit and then partially stages and commits as many times as commits you want to end up with. For example, suppose you want to split the middle commit of your three commits. Instead of “updated README formatting and added blame”, you want to split it into two commits: “updated README formatting” for the first, and “added blame” for the second. You can do that in the `rebase -i` script by changing the instruction on the commit you want to split to “edit”:

```
pick f7f3f6d changed my name a bit
edit 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

Then, when the script drops you to the command line, you reset that commit, take the changes that have been reset, and create multiple commits out of them. When you save and exit the editor, Git rewinds to the parent of the first commit in your list, applies the first commit (`f7f3f6d`), applies the second (`310154e`), and drops you to the console. There, you can do a mixed reset of that commit with `git reset HEAD^`, which effectively undoes that commit and leaves the modified files unstaged. Now you can stage and commit files until you have several commits, and run `git rebase --continue` when you’re done:

```
$ git reset HEAD^
$ git add README
$ git commit -m 'updated README formatting'
$ git add lib/simplegit.rb
$ git commit -m 'added blame'
$ git rebase --continue
```

Git applies the last commit (`a5f4a0d`) in the script, and your history looks like this:

```
$ git log -4 --pretty=format:"%h %s"
1c002dd added cat-file
9b29157 added blame
35cfb2b updated README formatting
f3cc40e changed my name a bit
```

Once again, this changes the SHA-1s of all the commits in your list, so make sure no commit shows up in that list that you've already pushed to a shared repository.

The Nuclear Option: filter-branch

There is another history-rewriting option that you can use if you need to rewrite a larger number of commits in some scriptable way – for instance, changing your e-mail address globally or removing a file from every commit. The command is `filter-branch`, and it can rewrite huge swaths of your history, so you probably shouldn't use it unless your project isn't yet public and other people haven't based work off the commits you're about to rewrite. However, it can be very useful. You'll learn a few of the common uses so you can get an idea of some of the things it's capable of.

REMOVING A FILE FROM EVERY COMMIT

This occurs fairly commonly. Someone accidentally commits a huge binary file with a thoughtless `git add .`, and you want to remove it everywhere. Perhaps you accidentally committed a file that contained a password, and you want to make your project open source. `filter-branch` is the tool you probably want to use to scrub your entire history. To remove a file named `passwords.txt` from your entire history, you can use the `--tree-filter` option to `filter-branch`:

```
$ git filter-branch --tree-filter 'rm -f passwords.txt' HEAD
Rewrite 6b9b3cf04e7c5686a9cb838c3f36a8cb6a0fc2bd (21/21)
Ref 'refs/heads/master' was rewritten
```

The `--tree-filter` option runs the specified command after each checkout of the project and then recommits the results. In this case, you remove a file called `passwords.txt` from every snapshot, whether it exists or not. If you want to remove all accidentally committed editor backup files, you can run something like `git filter-branch --tree-filter 'rm -f *~' HEAD`.

You'll be able to watch Git rewriting trees and commits and then move the branch pointer at the end. It's generally a good idea to do this in a testing branch and then hard-reset your master branch after you've determined the outcome is what you really want. To run `filter-branch` on all your branches, you can pass `--all` to the command.

MAKING A SUBDIRECTORY THE NEW ROOT

Suppose you've done an import from another source control system and have subdirectories that make no sense (trunk, tags, and so on). If you want to make the `trunk` subdirectory be the new project root for every commit, `filter-branch` can help you do that, too:

```
$ git filter-branch --subdirectory-filter trunk HEAD
Rewrite 856f0bf61e41a27326cd8e8f09fe708d679f596f (12/12)
Ref 'refs/heads/master' was rewritten
```

Now your new project root is what was in the `trunk` subdirectory each time. Git will also automatically remove commits that did not affect the subdirectory.

CHANGING E-MAIL ADDRESSES GLOBALLY

Another common case is that you forgot to run `git config` to set your name and e-mail address before you started working, or perhaps you want to open-source a project at work and change all your work e-mail addresses to your personal address. In any case, you can change e-mail addresses in multiple commits in a batch with `filter-branch` as well. You need to be careful to change only the e-mail addresses that are yours, so you use `--commit-filter`:

```
$ git filter-branch --commit-filter '
  if [ "$GIT_AUTHOR_EMAIL" = "schacon@localhost" ];
  then
    GIT_AUTHOR_NAME="Scott Chacon";
    GIT_AUTHOR_EMAIL="schacon@example.com";
    git commit-tree "$@";
  else
    git commit-tree "$@";
  fi' HEAD
```

This goes through and rewrites every commit to have your new address. Because commits contain the SHA-1 values of their parents, this command changes every commit SHA-1 in your history, not just those that have the matching e-mail address.

Reset Demystified

Before moving on to more specialized tools, let's talk about `reset` and `checkout`. These commands are two of the most confusing parts of Git when you first encounter them. They do so many things, that it seems hopeless to actually understand them and employ them properly. For this, we recommend a simple metaphor.

The Three Trees

An easier way to think about `reset` and `checkout` is through the mental frame of Git being a content manager of three different trees. By “tree” here we really mean “collection of files”, not specifically the data structure. (There are a few cases where the index doesn’t exactly act like a tree, but for our purposes it is easier to think about it this way for now.)

Git as a system manages and manipulates three trees in its normal operation:

| Tree | Role |
|-------------------|-----------------------------------|
| HEAD | Last commit snapshot, next parent |
| Index | Proposed next commit snapshot |
| Working Directory | Sandbox |

THE HEAD

HEAD is the pointer to the current branch reference, which is in turn a pointer to the last commit made on that branch. That means HEAD will be the parent of the next commit that is created. It’s generally simplest to think of HEAD as the snapshot of **your last commit**.

In fact, it’s pretty easy to see what that snapshot looks like. Here is an example of getting the actual directory listing and SHA-1 checksums for each file in the HEAD snapshot:

```
$ git cat-file -p HEAD
tree cfda3bf379e4f8dba8717dee55aab78aef7f4daf
author Scott Chacon 1301511835 -0700
committer Scott Chacon 1301511835 -0700
```

initial commit

```
$ git ls-tree -r HEAD
100644 blob a906cb2a4a904a152... README
100644 blob 8f94139338f9404f2... Rakefile
040000 tree 99f1a6d12cb4b6f19... lib
```

The `cat-file` and `ls-tree` commands are “plumbing” commands that are used for lower level things and not really used in day-to-day work, but they help us see what’s going on here.

THE INDEX

The Index is your **proposed next commit**. We’ve also been referring to this concept as Git’s “Staging Area” as this is what Git looks at when you run `git commit`.

Git populates this index with a list of all the file contents that were last checked out into your working directory and what they looked like when they were originally checked out. You then replace some of those files with new versions of them, and `git commit` converts that into the tree for a new commit.

```
$ git ls-files -s
100644 a906cb2a4a904a152e80877d4088654daad0c859 0      README
100644 8f94139338f9404f26296befa88755fc2598c289 0      Rakefile
100644 47c6340d6459e05787f644c2447d2595f5d3a54b 0      lib/simplegit.rb
```

Again, here we’re using `ls-files`, which is more of a behind the scenes command that shows you what your index currently looks like.

The index is not technically a tree structure – it’s actually implemented as a flattened manifest – but for our purposes it’s close enough.

THE WORKING DIRECTORY

Finally, you have your working directory. The other two trees store their content in an efficient but inconvenient manner, inside the `.git` folder. The Working Directory unpacks them into actual files, which makes it much easier for you to edit them. Think of the Working Directory as a **sandbox**, where you can try changes out before committing them to your staging area (index) and then to history.

```
$ tree
.
├── README
└── Rakefile
```

```
└─ lib  
    └─ simplegit.rb  
  
1 directory, 3 files
```

The Workflow

Git's main purpose is to record snapshots of your project in successively better states, by manipulating these three trees.

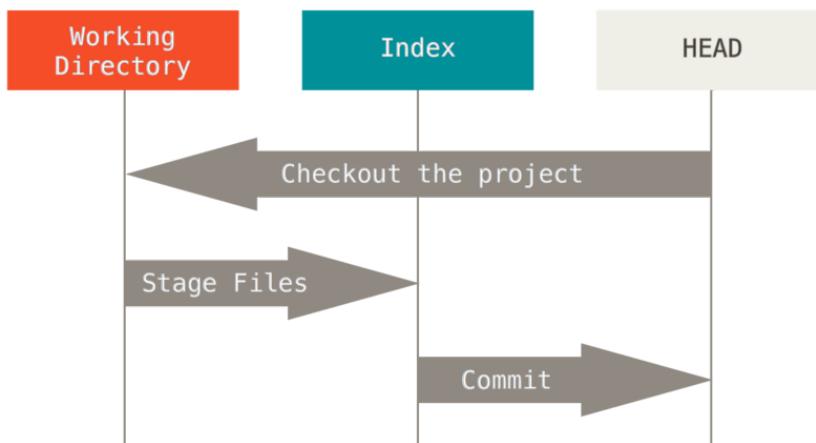
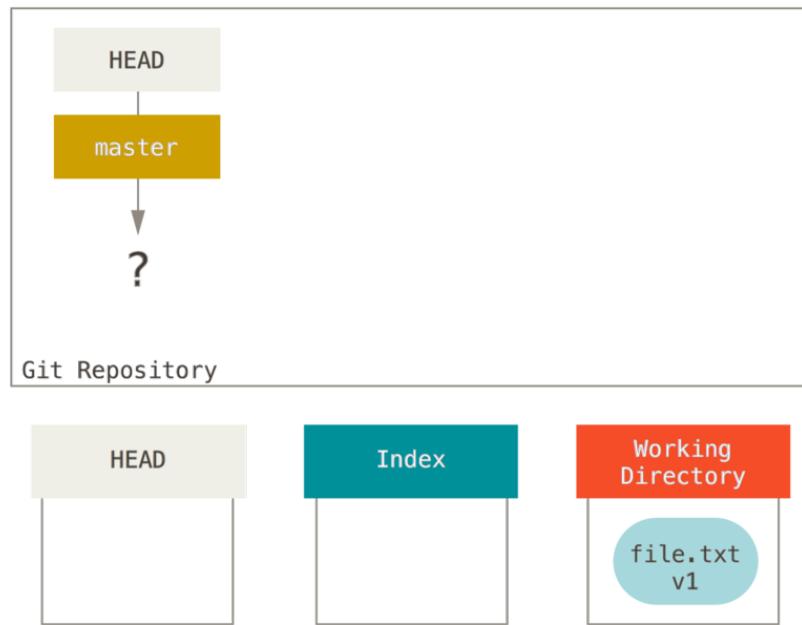


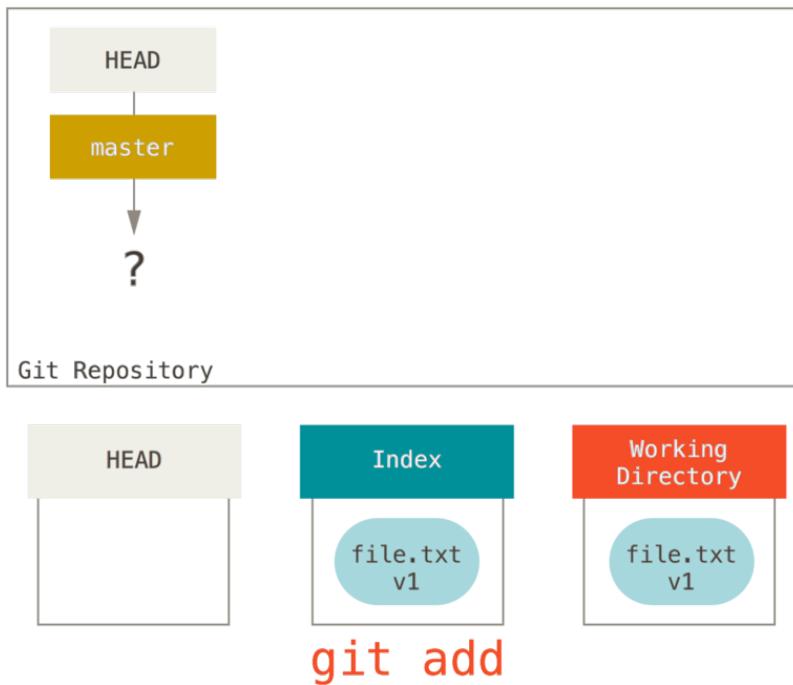
FIGURE 7-2

Let's visualize this process: say you go into a new directory with a single file in it. We'll call this **v1** of the file, and we'll indicate it in blue. Now we run `git init`, which will create a Git repository with a HEAD reference which points to an unborn branch (`master` doesn't exist yet).

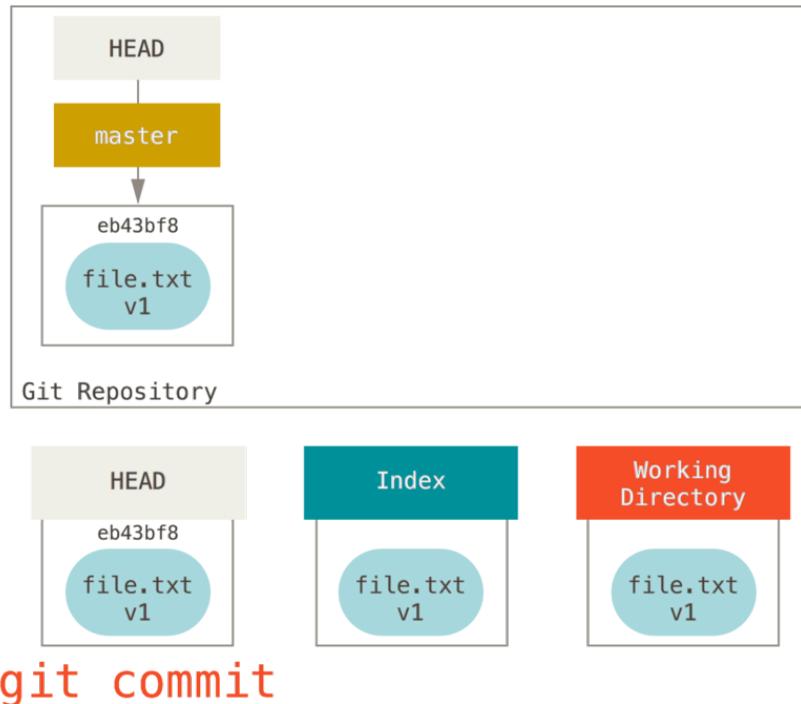
FIGURE 7-3

At this point, only the Working Directory tree has any content.

Now we want to commit this file, so we use `git add` to take content in the Working Directory and copy it to the Index.

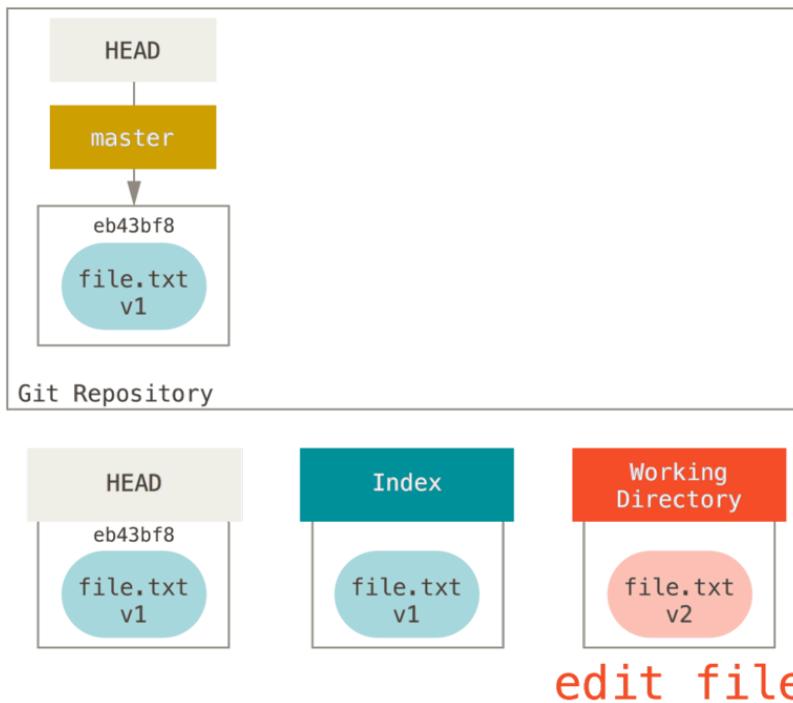
FIGURE 7-4

Then we run `git commit`, which takes the contents of the Index and saves it as a permanent snapshot, creates a commit object which points to that snapshot, and updates `master` to point to that commit.

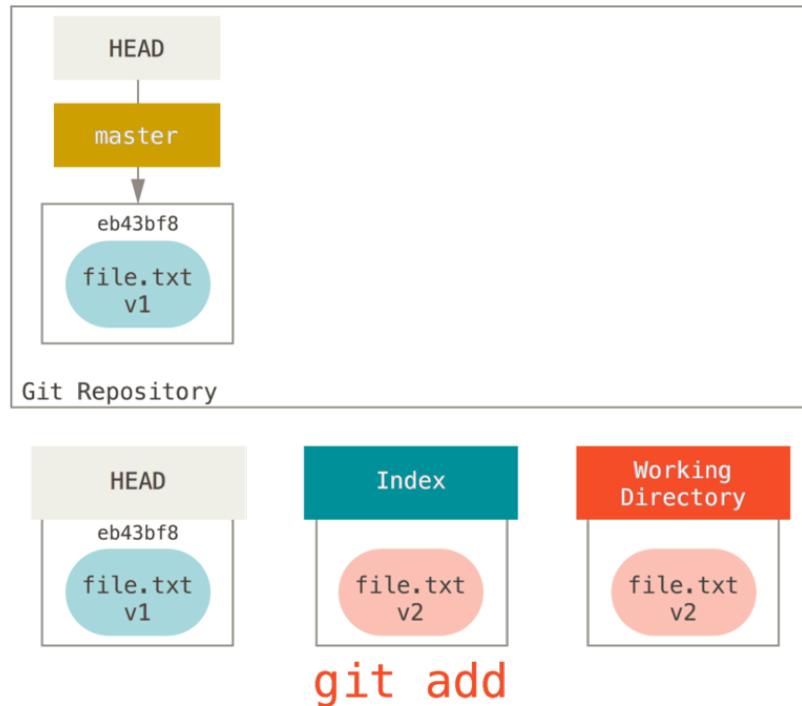
FIGURE 7-5

If we run `git status`, we'll see no changes, because all three trees are the same.

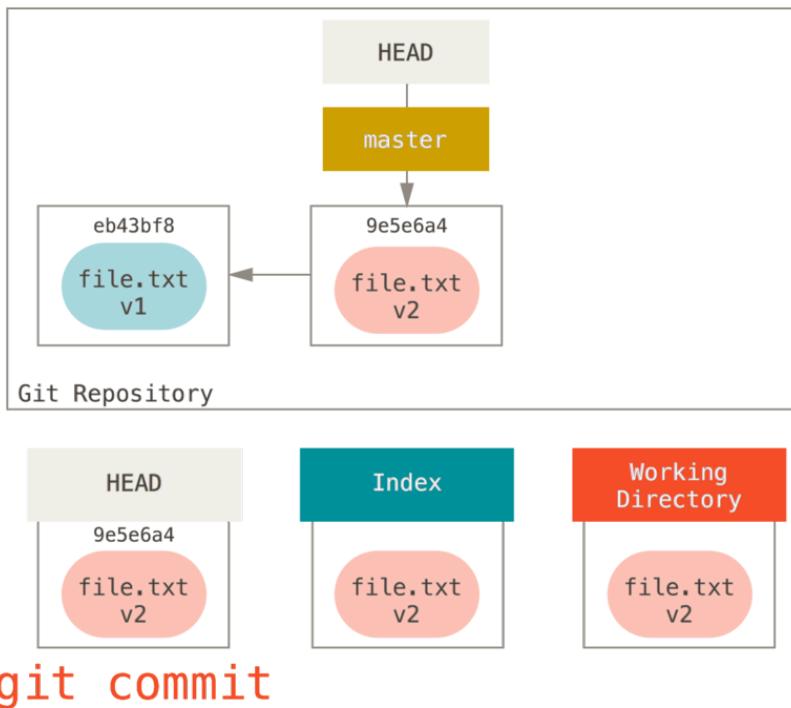
Now we want to make a change to that file and commit it. We'll go through the same process; first we change the file in our working directory. Let's call this **v2** of the file, and indicate it in red.

FIGURE 7-6

If we run `git status` right now, we'll see the file in red as "Changes not staged for commit," because that entry differs between the Index and the Working Directory. Next we run `git add` on it to stage it into our Index.

FIGURE 7-7

At this point if we run `git status` we will see the file in green under “Changes to be committed” because the Index and HEAD differ – that is, our proposed next commit is now different from our last commit. Finally, we run `git commit` to finalize the commit.

FIGURE 7-8

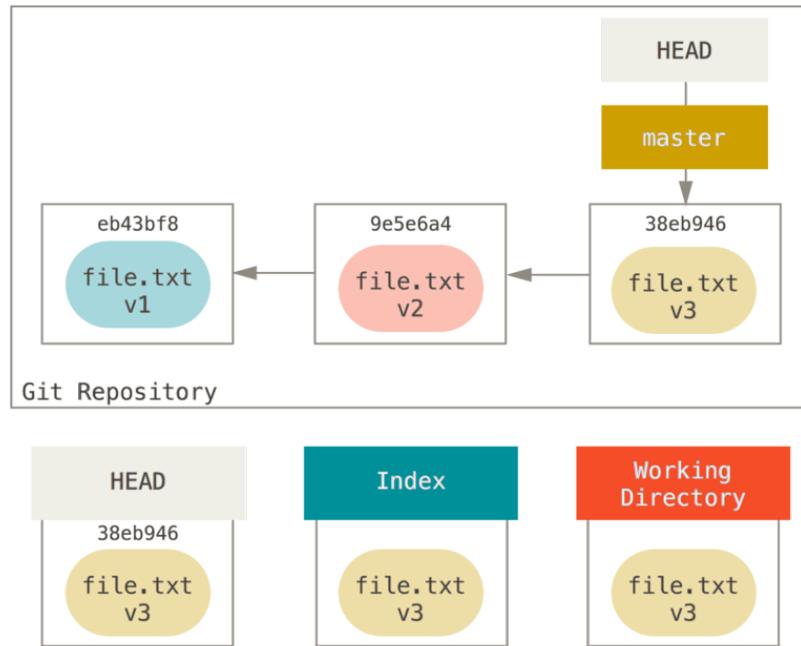
Now `git status` will give us no output, because all three trees are the same again.

Switching branches or cloning goes through a similar process. When you checkout a branch, it changes **HEAD** to point to the new branch ref, populates your **Index** with the snapshot of that commit, then copies the contents of the **Index** into your **Working Directory**.

The Role of Reset

The `reset` command makes more sense when viewed in this context.

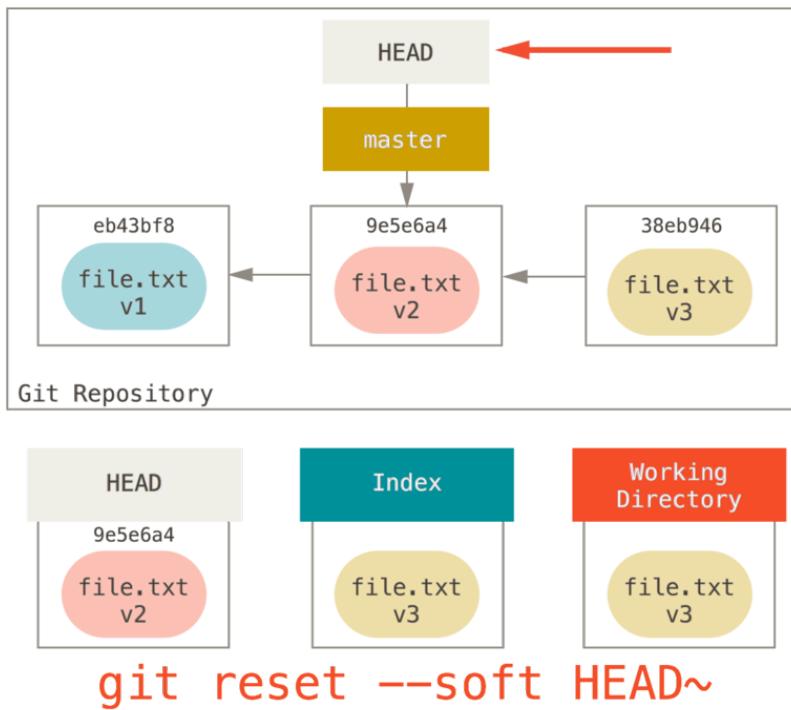
For the purposes of these examples, let's say that we've modified `file.txt` again and committed it a third time. So now our history looks like this:

FIGURE 7-9

Let's now walk through exactly what `reset` does when you call it. It directly manipulates these three trees in a simple and predictable way. It does up to three basic operations.

STEP 1: MOVE HEAD

The first thing `reset` will do is move what HEAD points to. This isn't the same as changing HEAD itself (which is what `checkout` does); `reset` moves the branch that HEAD is pointing to. This means if HEAD is set to the `master` branch (i.e. you're currently on the `master` branch), running `git reset 9e5e64a` will start by making `master` point to `9e5e64a`.

FIGURE 7-10

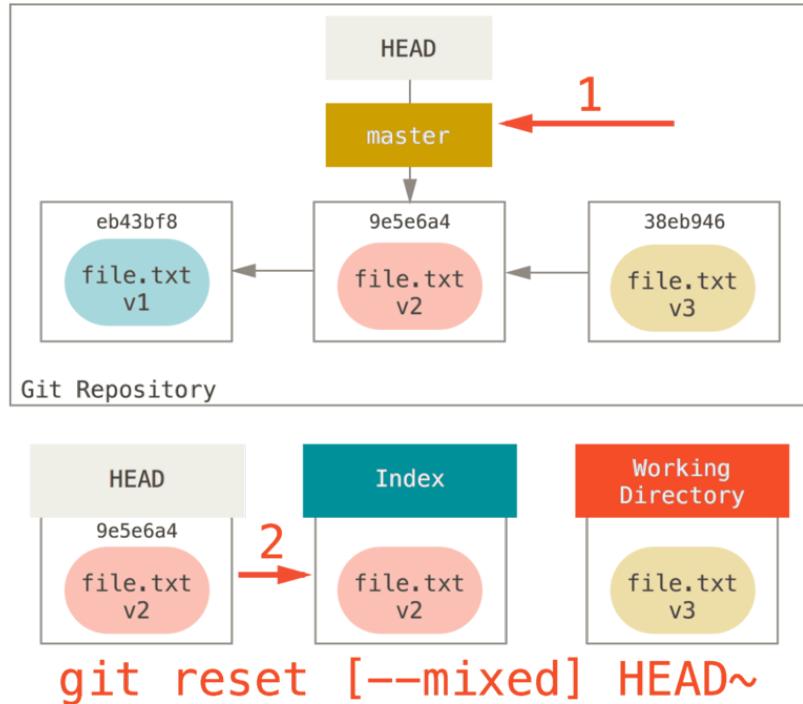
No matter what form of `reset` with a commit you invoke, this is the first thing it will always try to do. With `reset --soft`, it will simply stop there.

Now take a second to look at that diagram and realize what happened: it essentially undid the last `git commit` command. When you run `git commit`, Git creates a new commit and moves the branch that `HEAD` points to up to it. When you `reset` back to `HEAD~` (the parent of `HEAD`), you are moving the branch back to where it was, without changing the Index or Working Directory. You could now update the Index and run `git commit` again to accomplish what `git commit --amend` would have done (see “[Changing the Last Commit](#)”).

STEP 2: UPDATING THE INDEX (--MIXED)

Note that if you run `git status` now you’ll see in green the difference between the Index and what the new `HEAD` is.

The next thing `reset` will do is to update the Index with the contents of whatever snapshot `HEAD` now points to.

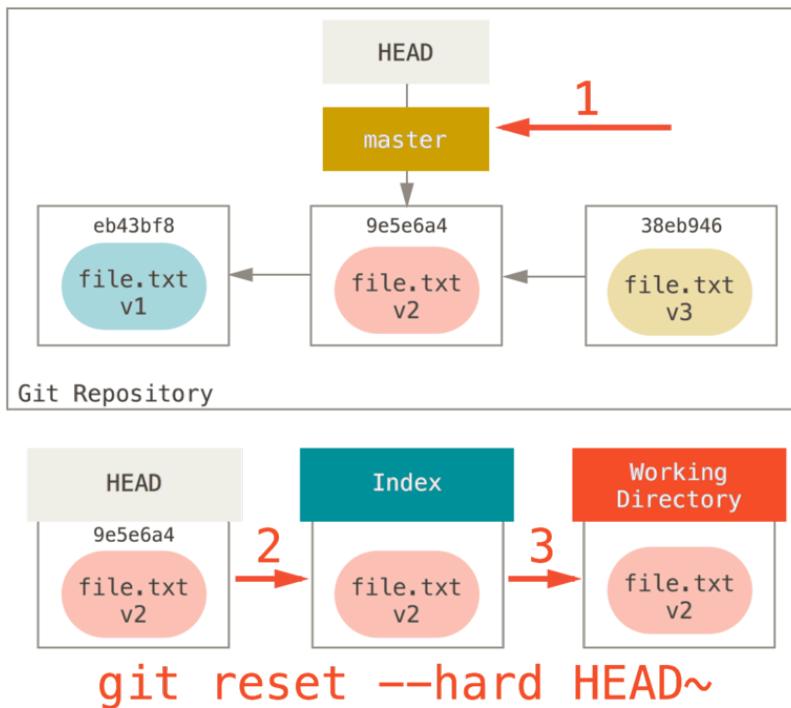
FIGURE 7-11

If you specify the `--mixed` option, `reset` will stop at this point. This is also the default, so if you specify no option at all (just `git reset HEAD~` in this case), this is where the command will stop.

Now take another second to look at that diagram and realize what happened: it still undid your last `commit`, but also *unstaged* everything. You rolled back to before you ran all your `git add` and `git commit` commands.

STEP 3: UPDATING THE WORKING DIRECTORY (--HARD)

The third thing that `reset` will do is to make the Working Directory look like the Index. If you use the `--hard` option, it will continue to this stage.

FIGURE 7-12

So let's think about what just happened. You undid your last commit, the `git add` and `git commit` commands, **and** all the work you did in your working directory.

It's important to note that this flag (`--hard`) is the only way to make the `reset` command dangerous, and one of the very few cases where Git will actually destroy data. Any other invocation of `reset` can be pretty easily undone, but the `--hard` option cannot, since it forcibly overwrites files in the Working Directory. In this particular case, we still have the **v3** version of our file in a commit in our Git DB, and we could get it back by looking at our `reflog`, but if we had not committed it, Git still would have overwritten the file and it would be unrecoverable.

RECAP

The `reset` command overwrites these three trees in a specific order, stopping when you tell it to:

1. Move the branch `HEAD` points to (*stop here if --soft*)

2. Make the Index look like HEAD (*stop here unless --hard*)
3. Make the Working Directory look like the Index

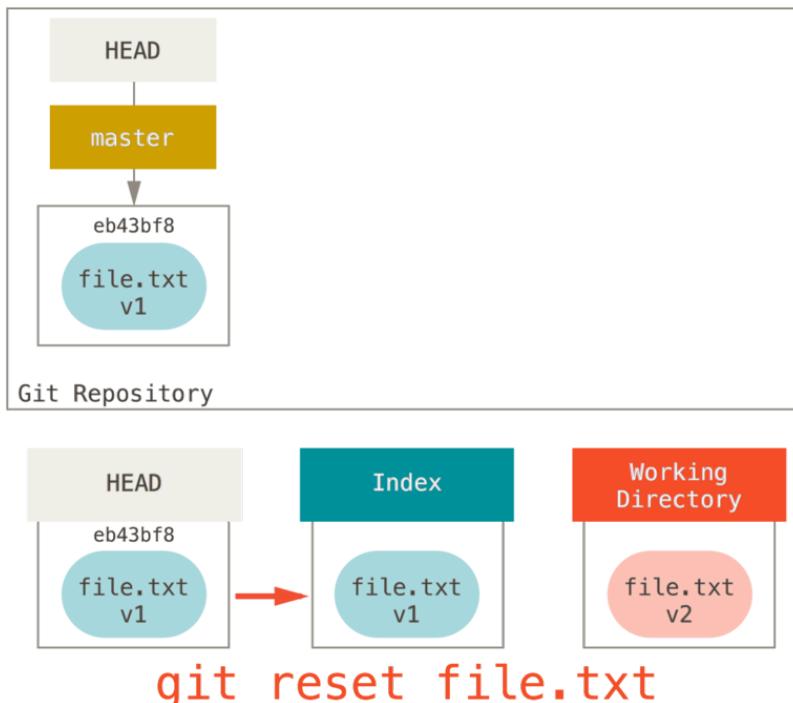
Reset With a Path

That covers the behavior of `reset` in its basic form, but you can also provide it with a path to act upon. If you specify a path, `reset` will skip step 1, and limit the remainder of its actions to a specific file or set of files. This actually sort of makes sense – HEAD is just a pointer, and you can't point to part of one commit and part of another. But the Index and Working directory *can* be partially updated, so `reset` proceeds with steps 2 and 3.

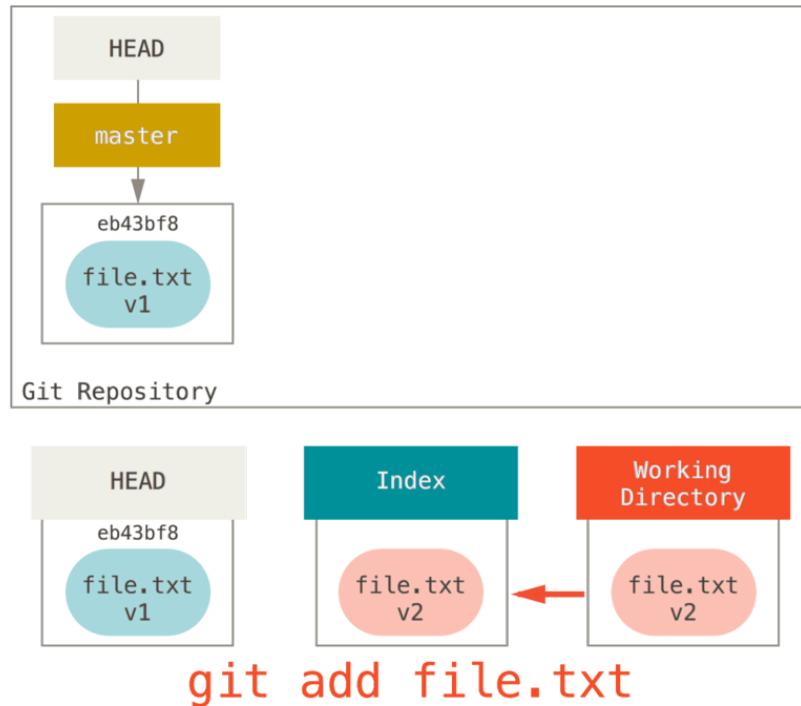
So, assume we run `git reset file.txt`. This form (since you did not specify a commit SHA-1 or branch, and you didn't specify `--soft` or `--hard`) is shorthand for `git reset --mixed HEAD file.txt`, which will:

1. Move the branch HEAD points to (*skipped*)
2. Make the Index look like HEAD (*stop here*)

So it essentially just copies `file.txt` from HEAD to the Index.

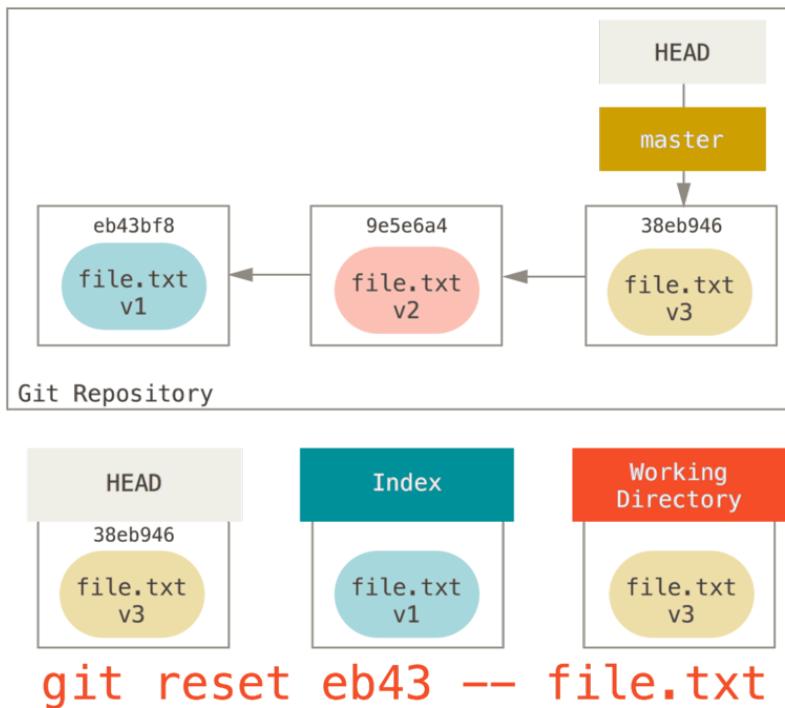
FIGURE 7-13

This has the practical effect of *unstaging* the file. If we look at the diagram for that command and think about what `git add` does, they are exact opposites.

FIGURE 7-14

This is why the output of the `git status` command suggests that you run this to unstage a file. (See “**Deshacer un Archivo Preparado**” for more on this.)

We could just as easily not let Git assume we meant “pull the data from HEAD” by specifying a specific commit to pull that file version from. We would just run something like `git reset eb43bf file.txt`.

FIGURE 7-15

This effectively does the same thing as if we had reverted the content of the file to **v1** in the Working Directory, ran `git add` on it, then reverted it back to **v3** again (without actually going through all those steps). If we run `git commit` now, it will record a change that reverts that file back to **v1**, even though we never actually had it in our Working Directory again.

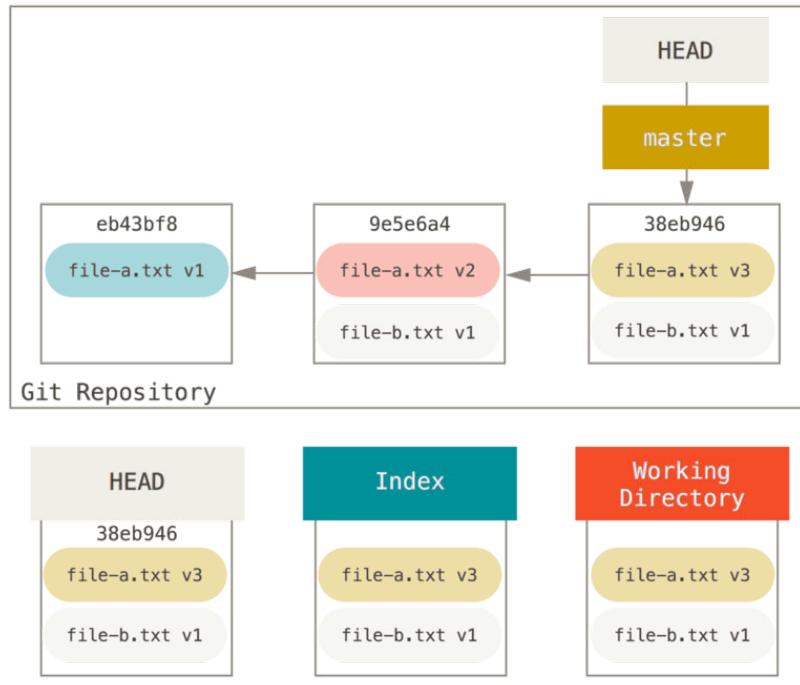
It's also interesting to note that like `git add`, the `reset` command will accept a `--patch` option to unstage content on a hunk-by-hunk basis. So you can selectively unstage or revert content.

Squashing

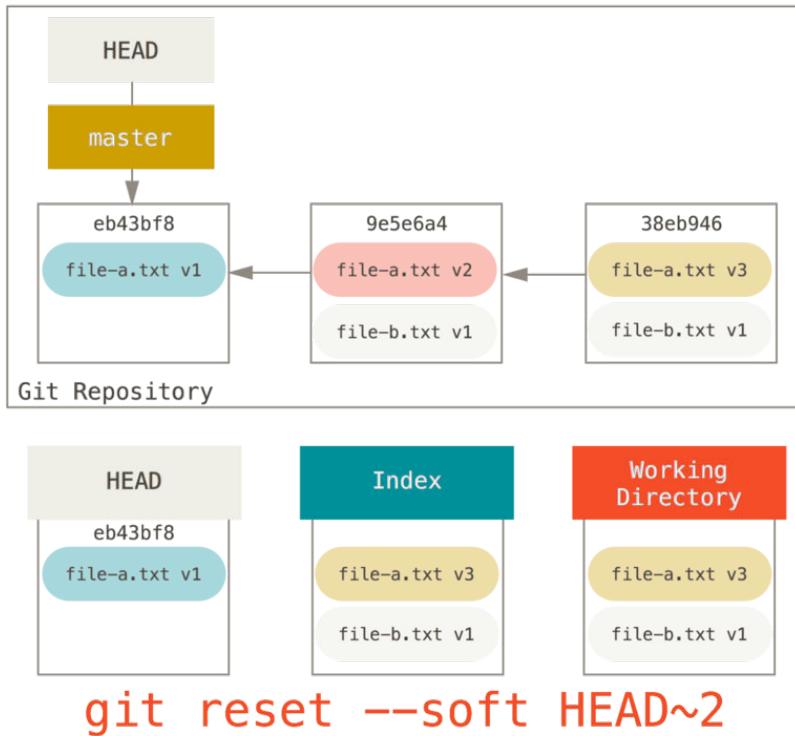
Let's look at how to do something interesting with this newfound power – squashing commits.

Say you have a series of commits with messages like “oops.”, “WIP” and “forgot this file”. You can use `reset` to quickly and easily squash them into a single commit that makes you look really smart. (“**Squashing Commits**” shows another way to do this, but in this example it’s simpler to use `reset`.)

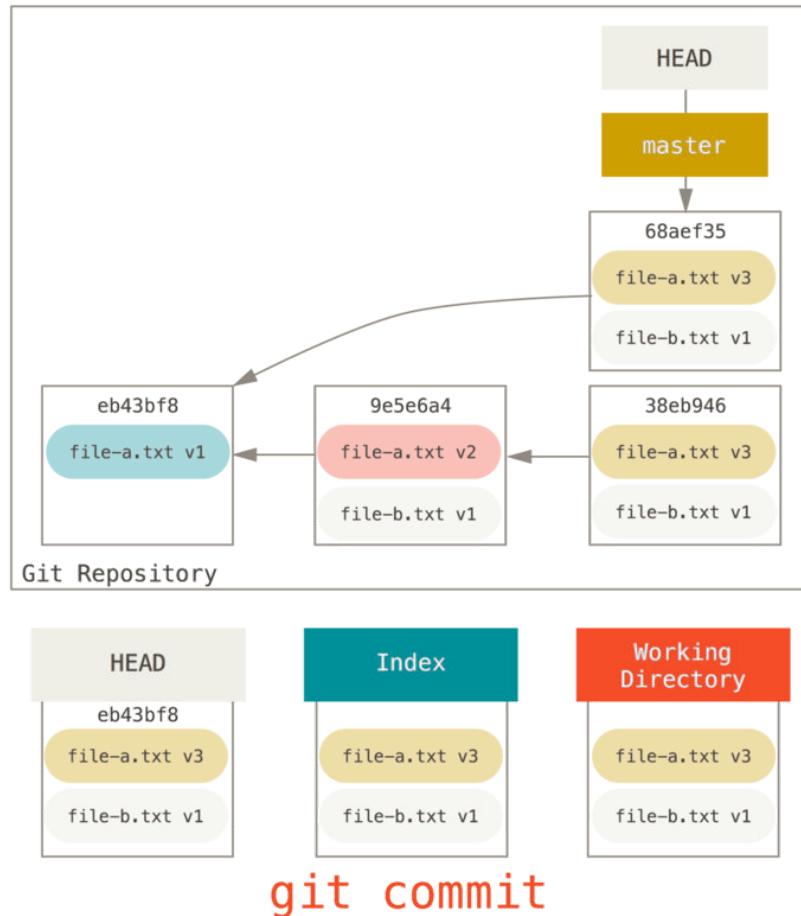
Let's say you have a project where the first commit has one file, the second commit added a new file and changed the first, and the third commit changed the first file again. The second commit was a work in progress and you want to squash it down.

FIGURE 7-16

You can run `git reset --soft HEAD~2` to move the HEAD branch back to an older commit (the first commit you want to keep):

FIGURE 7-17

And then simply run `git commit` again:

FIGURE 7-18

Now you can see that your reachable history, the history you would push, now looks like you had one commit with `file-a.txt v1`, then a second that both modified `file-a.txt` to `v3` and added `file-b.txt`. The commit with the `v2` version of the file is no longer in the history.

Check It Out

Finally, you may wonder what the difference between `checkout` and `reset` is. Like `reset`, `checkout` manipulates the three trees, and it is a bit different depending on whether you give the command a file path or not.

WITHOUT PATHS

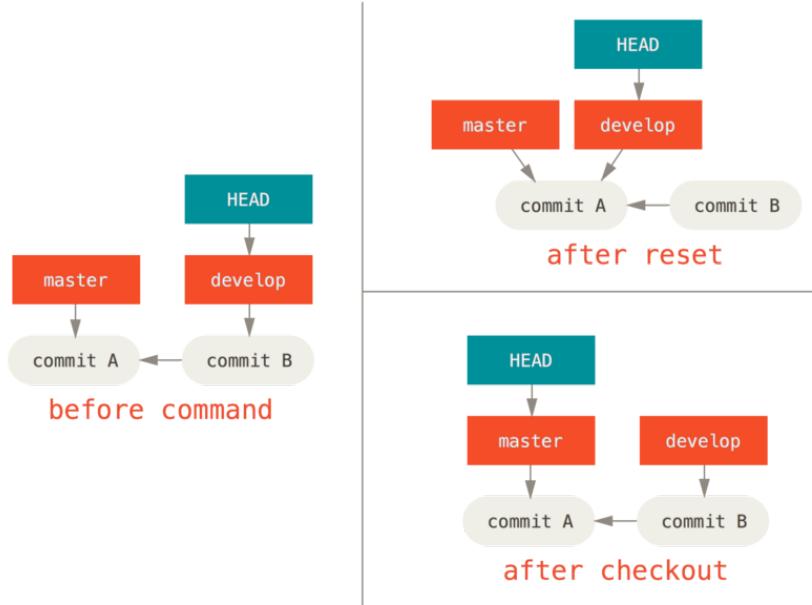
Running `git checkout [branch]` is pretty similar to running `git reset --hard [branch]` in that it updates all three trees for you to look like `[branch]`, but there are two important differences.

First, unlike `reset --hard`, `checkout` is working-directory safe; it will check to make sure it's not blowing away files that have changes to them. Actually, it's a bit smarter than that – it tries to do a trivial merge in the Working Directory, so all of the files you *haven't* changed in will be updated. `reset --hard`, on the other hand, will simply replace everything across the board without checking.

The second important difference is how it updates HEAD. Where `reset` will move the branch that HEAD points to, `checkout` will move HEAD itself to point to another branch.

For instance, say we have `master` and `develop` branches which point at different commits, and we're currently on `develop` (so HEAD points to it). If we run `git reset master`, `develop` itself will now point to the same commit that `master` does. If we instead run `git checkout master`, `develop` does not move, HEAD itself does. HEAD will now point to `master`.

So, in both cases we're moving HEAD to point to commit A, but *how* we do so is very different. `reset` will move the branch HEAD points to, `checkout` moves HEAD itself.

FIGURE 7-19

WITH PATHS

The other way to run `checkout` is with a file path, which, like `reset`, does not move `HEAD`. It is just like `git reset [branch] file` in that it updates the index with that file at that commit, but it also overwrites the file in the working directory. It would be exactly like `git reset --hard [branch] file` (if `reset` would let you run that) – it's not working-directory safe, and it does not move `HEAD`.

Also, like `git reset` and `git add`, `checkout` will accept a `--patch` option to allow you to selectively revert file contents on a hunk-by-hunk basis.

Summary

Hopefully now you understand and feel more comfortable with the `reset` command, but are probably still a little confused about how exactly it differs from `checkout` and could not possibly remember all the rules of the different invocations.

Here's a cheat-sheet for which commands affect which trees. The "HEAD" column reads "REF" if that command moves the reference (branch) that `HEAD` points to, and "HEAD" if it moves `HEAD` itself. Pay especial attention to the `WD`

Safe? column – if it says **NO**, take a second to think before running that command.

| | | HEAD | Index | Workdir | WD Safe? |
|---------------------------------------|------|------|-------|-----------|----------|
| Commit Level | | | | | |
| <code>reset --soft [commit]</code> | REF | NO | NO | YES | |
| <code>reset [commit]</code> | REF | YES | NO | YES | |
| <code>reset --hard [commit]</code> | REF | YES | YES | NO | |
| <code>checkout [commit]</code> | HEAD | YES | YES | YES | |
| File Level | | | | | |
| <code>reset (commit) [file]</code> | NO | YES | NO | YES | |
| <code>checkout (commit) [file]</code> | NO | YES | YES | NO | |

Advanced Merging

Merging in Git is typically fairly easy. Since Git makes it easy to merge another branch multiple times, it means that you can have a very long lived branch but you can keep it up to date as you go, solving small conflicts often, rather than be surprised by one enormous conflict at the end of the series.

However, sometimes tricky conflicts do occur. Unlike some other version control systems, Git does not try to be overly clever about merge conflict resolution. Git's philosophy is to be smart about determining when a merge resolution is unambiguous, but if there is a conflict, it does not try to be clever about automatically resolving it. Therefore, if you wait too long to merge two branches that diverge quickly, you can run into some issues.

In this section, we'll go over what some of those issues might be and what tools Git gives you to help handle these more tricky situations. We'll also cover some of the different, non-standard types of merges you can do, as well as see how to back out of merges that you've done.

Merge Conflicts

While we covered some basics on resolving merge conflicts in “**Principales Conflictos que Pueden Surgir en las Fusiones**”, for more complex conflicts, Git provides a few tools to help you figure out what's going on and how to better deal with the conflict.

First of all, if at all possible, try to make sure your working directory is clean before doing a merge that may have conflicts. If you have work in progress, either commit it to a temporary branch or stash it. This makes it so that you can undo **anything** you try here. If you have unsaved changes in your working directory when you try a merge, some of these tips may help you lose that work.

Let's walk through a very simple example. We have a super simple Ruby file that prints *hello world*.

```
#!/usr/bin/env ruby

def hello
    puts 'hello world'
end

hello()
```

In our repository, we create a new branch named `whitespace` and proceed to change all the Unix line endings to DOS line endings, essentially changing every line of the file, but just with whitespace. Then we change the line “hello world” to “hello mundo”.

```
$ git checkout -b whitespace
Switched to a new branch 'whitespace'

$ unix2dos hello.rb
unix2dos: converting file hello.rb to DOS format ...
$ git commit -am 'converted hello.rb to DOS'
[whitespace 3270f76] converted hello.rb to DOS
 1 file changed, 7 insertions(+), 7 deletions(-)

$ vim hello.rb
$ git diff -w
diff --git a/hello.rb b/hello.rb
index ac51efd..e85207e 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,7 +1,7 @@
 #! /usr/bin/env ruby

 def hello
- puts 'hello world'
+ puts 'hello mundo'^M
end

hello()

$ git commit -am 'hello mundo change'
```

```
[whitespace 6d338d2] hello mundo change
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Now we switch back to our `master` branch and add some documentation for the function.

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

$ vim hello.rb
$ git diff
diff --git a/hello.rb b/hello.rb
index ac51efd..36c06c8 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,5 +1,6 @@
 #! /usr/bin/env ruby

+## prints out a greeting
def hello
    puts 'hello world'
end

$ git commit -am 'document the function'
[master bec6336] document the function
 1 file changed, 1 insertion(+)
```

Now we try to merge in our `whitespace` branch and we'll get conflicts because of the whitespace changes.

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

ABORTING A MERGE

We now have a few options. First, let's cover how to get out of this situation. If you perhaps weren't expecting conflicts and don't want to quite deal with the situation yet, you can simply back out of the merge with `git merge --abort`.

```
$ git status -sb
## master
```

```
UU hello.rb

$ git merge --abort

$ git status -sb
## master
```

The `git merge --abort` option tries to revert back to your state before you ran the merge. The only cases where it may not be able to do this perfectly would be if you had unstashed, uncommitted changes in your working directory when you ran it, otherwise it should work fine.

If for some reason you find yourself in a horrible state and just want to start over, you can also run `git reset --hard HEAD` or wherever you want to get back to. Remember again that this will blow away your working directory, so make sure you don't want any changes there.

IGNORING WHITESPACE

In this specific case, the conflicts are whitespace related. We know this because the case is simple, but it's also pretty easy to tell in real cases when looking at the conflict because every line is removed on one side and added again on the other. By default, Git sees all of these lines as being changed, so it can't merge the files.

The default merge strategy can take arguments though, and a few of them are about properly ignoring whitespace changes. If you see that you have a lot of whitespace issues in a merge, you can simply abort it and do it again, this time with `-Xignore-all-space` or `-Xignore-space-change`. The first option ignores changes in any **amount** of existing whitespace, the second ignores all whitespace changes altogether.

```
$ git merge -Xignore-all-space whitespace
Auto-merging hello.rb
Merge made by the 'recursive' strategy.
hello.rb | 2 ++
1 file changed, 1 insertion(+), 1 deletion(-)
```

Since in this case, the actual file changes were not conflicting, once we ignore the whitespace changes, everything merges just fine.

This is a lifesaver if you have someone on your team who likes to occasionally reformat everything from spaces to tabs or vice-versa.

MANUAL FILE RE-MERGING

Though Git handles whitespace pre-processing pretty well, there are other types of changes that perhaps Git can't handle automatically, but are scriptable fixes. As an example, let's pretend that Git could not handle the whitespace change and we needed to do it by hand.

What we really need to do is run the file we're trying to merge in through a dos2unix program before trying the actual file merge. So how would we do that?

First, we get into the merge conflict state. Then we want to get copies of my version of the file, their version (from the branch we're merging in) and the common version (from where both sides branched off). Then we want to fix up either their side or our side and re-try the merge again for just this single file.

Getting the three file versions is actually pretty easy. Git stores all of these versions in the index under “stages” which each have numbers associated with them. Stage 1 is the common ancestor, stage 2 is your version and stage 3 is from the MERGE_HEAD, the version you’re merging in (“theirs”).

You can extract a copy of each of these versions of the conflicted file with the `git show` command and a special syntax.

```
$ git show :1:hello.rb > hello.common.rb
$ git show :2:hello.rb > hello.ours.rb
$ git show :3:hello.rb > hello.theirs.rb
```

If you want to get a little more hard core, you can also use the `ls-files -u` plumbing command to get the actual SHA-1s of the Git blobs for each of these files.

```
$ git ls-files -u
100755 ac51efdc3df4f4fd328d1a02ad05331d8e2c9111 1      hello.rb
100755 36c06c8752c78d2aff89571132f3bf7841a7b5c3 2      hello.rb
100755 e85207e04dfdd5eb0a1e9febbc67fd837c44a1cd 3      hello.rb
```

The `:1:hello.rb` is just a shorthand for looking up that blob SHA-1.

Now that we have the content of all three stages in our working directory, we can manually fix up theirs to fix the whitespace issue and re-merge the file with the little-known `git merge-file` command which does just that.

```
$ dos2unix hello.theirs.rb
dos2unix: converting file hello.theirs.rb to Unix format ...
```

```
$ git merge-file -p \
    hello.ours.rb hello.common.rb hello.theirs.rb > hello.rb

$ git diff -w
diff --cc hello.rb
index 36c06c8,e85207e..0000000
--- a/hello.rb
+++ b/hello.rb
@@@ -1,8 -1,7 +1,8 @@@
  #! /usr/bin/env ruby

  +# prints out a greeting
  def hello
-  puts 'hello world'
+  puts 'hello mundo'
  end

  hello()
```

At this point we have nicely merged the file. In fact, this actually works better than the `ignore-all-space` option because this actually fixes the whitespace changes before merge instead of simply ignoring them. In the `ignore-all-space` merge, we actually ended up with a few lines with DOS line endings, making things mixed.

If you want to get an idea before finalizing this commit about what was actually changed between one side or the other, you can ask `git diff` to compare what is in your working directory that you're about to commit as the result of the merge to any of these stages. Let's go through them all.

To compare your result to what you had in your branch before the merge, in other words, to see what the merge introduced, you can run `git diff --ours`

```
$ git diff --ours
* Unmerged path hello.rb
diff --git a/hello.rb b/hello.rb
index 36c06c8..44d0a25 100755
--- a/hello.rb
+++ b/hello.rb
@@ -2,7 +2,7 @@
  # prints out a greeting
  def hello
-  puts 'hello world'
+  puts 'hello mundo'
  end
```

```
hello()
```

So here we can easily see that what happened in our branch, what we're actually introducing to this file with this merge, is changing that single line.

If we want to see how the result of the merge differed from what was on their side, you can run `git diff --theirs`. In this and the following example, we have to use `-w` to strip out the whitespace because we're comparing it to what is in Git, not our cleaned up `hello.theirs.rb` file.

```
$ git diff --theirs -w
* Unmerged path hello.rb
diff --git a/hello.rb b/hello.rb
index e85207e..44d0a25 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,5 +1,6 @@
 #! /usr/bin/env ruby

+# prints out a greeting
def hello
    puts 'hello mundo'
end
```

Finally, you can see how the file has changed from both sides with `git diff --base`.

```
$ git diff --base -w
* Unmerged path hello.rb
diff --git a/hello.rb b/hello.rb
index ac51efd..44d0a25 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,7 +1,8 @@
 #! /usr/bin/env ruby

+# prints out a greeting
def hello
- puts 'hello world'
+ puts 'hello mundo'
end

hello()
```

At this point we can use the `git clean` command to clear out the extra files we created to do the manual merge but no longer need.

```
$ git clean -f
Removing hello.common.rb
Removing hello.ours.rb
Removing hello.theirs.rb
```

CHECKING OUT CONFLICTS

Perhaps we're not happy with the resolution at this point for some reason, or maybe manually editing one or both sides still didn't work well and we need more context.

Let's change up the example a little. For this example, we have two longer lived branches that each have a few commits in them but create a legitimate content conflict when merged.

```
$ git log --graph --oneline --decorate --all
* f1270f7 (HEAD, master) update README
* 9af9d3b add a README
* 694971d update phrase to hola world
| * e3eb223 (mundo) add more tests
| * 7cff591 add testing script
| * c3ffff1 changed text to hello mundo
|/
* b7dcc89 initial hello world code
```

We now have three unique commits that live only on the `master` branch and three others that live on the `mundo` branch. If we try to merge the `mundo` branch in, we get a conflict.

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

We would like to see what the merge conflict is. If we open up the file, we'll see something like this:

```
#!/usr/bin/env ruby

def hello
```

```
<<<<< HEAD
 puts 'hola mundo'
=====
 puts 'hello mundo'
>>>>> mundo
end

hello()
```

Both sides of the merge added content to this file, but some of the commits modified the file in the same place that caused this conflict.

Let's explore a couple of tools that you now have at your disposal to determine how this conflict came to be. Perhaps it's not obvious how exactly you should fix this conflict. You need more context.

One helpful tool is `git checkout` with the ‘--conflict’ option. This will re-checkout the file again and replace the merge conflict markers. This can be useful if you want to reset the markers and try to resolve them again.

You can pass `--conflict` either `diff3` or `merge` (which is the default). If you pass it `diff3`, Git will use a slightly different version of conflict markers, not only giving you the “ours” and “theirs” versions, but also the “base” version inline to give you more context.

```
$ git checkout --conflict=diff3 hello.rb
```

Once we run that, the file will look like this instead:

```
#!/usr/bin/env ruby

def hello
<<<<< ours
 puts 'hola mundo'
||||||| base
 puts 'hello world'
=====
 puts 'hello mundo'
>>>>> theirs
end

hello()
```

If you like this format, you can set it as the default for future merge conflicts by setting the `merge.conflictstyle` setting to `diff3`.

```
$ git config --global merge.conflictstyle diff3
```

The `git checkout` command can also take `--ours` and `--theirs` options, which can be a really fast way of just choosing either one side or the other without merging things at all.

This can be particularly useful for conflicts of binary files where you can simply choose one side, or where you only want to merge certain files in from another branch - you can do the merge and then checkout certain files from one side or the other before committing.

MERGE LOG

Another useful tool when resolving merge conflicts is `git log`. This can help you get context on what may have contributed to the conflicts. Reviewing a little bit of history to remember why two lines of development were touching the same area of code can be really helpful sometimes.

To get a full list of all of the unique commits that were included in either branch involved in this merge, we can use the “triple dot” syntax that we learned in “[Triple Dot](#)”.

```
$ git log --oneline --left-right HEAD...MERGE_HEAD
< f1270f7 update README
< 9af9d3b add a README
< 694971d update phrase to hola world
> e3eb223 add more tests
> 7cff591 add testing script
> c3ffff1 changed text to hello mundo
```

That’s a nice list of the six total commits involved, as well as which line of development each commit was on.

We can further simplify this though to give us much more specific context. If we add the `--merge` option to `git log`, it will only show the commits in either side of the merge that touch a file that’s currently conflicted.

```
$ git log --oneline --left-right --merge
< 694971d update phrase to hola world
> c3ffff1 changed text to hello mundo
```

If you run that with the `-p` option instead, you get just the diffs to the file that ended up in conflict. This can be **really** helpful in quickly giving you the

context you need to help understand why something conflicts and how to more intelligently resolve it.

COMBINED DIFF FORMAT

Since Git stages any merge results that are successful, when you run `git diff` while in a conflicted merge state, you only get what is currently still in conflict. This can be helpful to see what you still have to resolve.

When you run `git diff` directly after a merge conflict, it will give you information in a rather unique diff output format.

```
$ git diff
diff --cc hello.rb
index 0399cd5,59727f0..0000000
--- a/hello.rb
+++ b/hello.rb
@@@ -1,7 -1,7 +1,11 @@
 #! /usr/bin/env ruby

def hello
<<<<<< HEAD
+ puts 'hola mundo'
=====
+ puts 'hello mundo'
++>>>>> mundo
end

hello()
```

The format is called “Combined Diff” and gives you two columns of data next to each line. The first column shows you if that line is different (added or removed) between the “ours” branch and the file in your working directory and the second column does the same between the “theirs” branch and your working directory copy.

So in that example you can see that the `<<<<<` and `>>>>>` lines are in the working copy but were not in either side of the merge. This makes sense because the merge tool stuck them in there for our context, but we’re expected to remove them.

If we resolve the conflict and run `git diff` again, we’ll see the same thing, but it’s a little more useful.

```
$ vim hello.rb
$ git diff
```

```

diff --cc hello.rb
index 0399cd5,59727f0..0000000
--- a/hello.rb
+++ b/hello.rb
@@@ -1,7 -1,7 +1,7 @@@
  #! /usr/bin/env ruby

  def hello
-   puts 'hola world'
-   puts 'hello mundo'
++ puts 'hola mundo'
  end

  hello()

```

This shows us that “hola world” was in our side but not in the working copy, that “hello mundo” was in their side but not in the working copy and finally that “hola mundo” was not in either side but is now in the working copy. This can be useful to review before committing the resolution.

You can also get this from the `git log` for any merge after the fact to see how something was resolved after the fact. Git will output this format if you run `git show` on a merge commit, or if you add a `--cc` option to a `git log -p` (which by default only shows patches for non-merge commits).

```

$ git log --cc -p -1
commit 14f41939956d80b9e17bb8721354c33f8d5b5a79
Merge: f1270f7 e3eb223
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Sep 19 18:14:49 2014 +0200

    Merge branch 'mundo'

    Conflicts:
        hello.rb

diff --cc hello.rb
index 0399cd5,59727f0..e1d0799
--- a/hello.rb
+++ b/hello.rb
@@@ -1,7 -1,7 +1,7 @@@
  #! /usr/bin/env ruby

  def hello
-   puts 'hola world'
-   puts 'hello mundo'
++ puts 'hola mundo'
  end

```

```
hello()
```

Undoing Merges

Now that you know how to create a merge commit, you'll probably make some by mistake. One of the great things about working with Git is that it's okay to make mistakes, because it's possible (and in many cases easy) to fix them.

Merge commits are no different. Let's say you started work on a topic branch, accidentally merged it into `master`, and now your commit history looks like this:

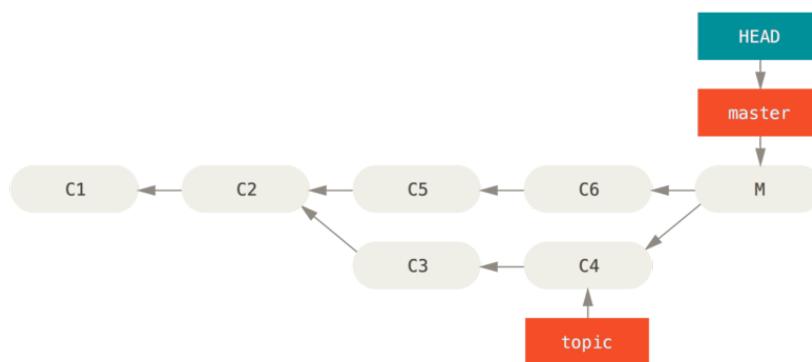


FIGURE 7-20
Accidental merge commit

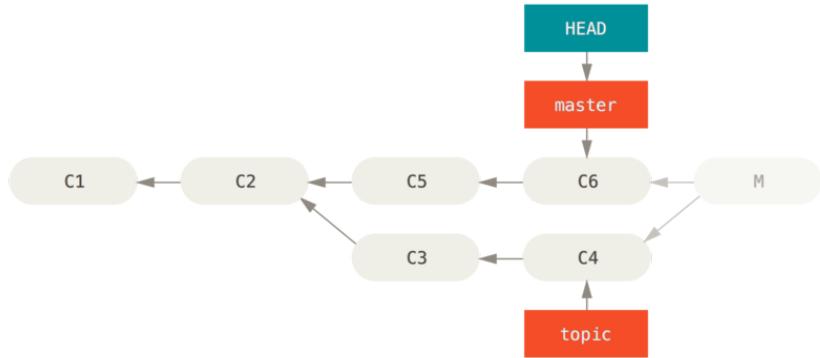
There are two ways to approach this problem, depending on what your desired outcome is.

FIX THE REFERENCES

If the unwanted merge commit only exists on your local repository, the easiest and best solution is to move the branches so that they point where you want them to. In most cases, if you follow the errant `git merge` with `git reset --hard HEAD~`, this will reset the branch pointers so they look like this:

FIGURE 7-21

*History after git
reset --hard
HEAD~*



We covered `reset` back in “[Reset Demystified](#)”, so it shouldn’t be too hard to figure out what’s going on here. Here’s a quick refresher: `reset --hard` usually goes through three steps:

1. Move the branch `HEAD` points to. In this case, we want to move `master` to where it was before the merge commit (`C6`).
2. Make the index look like `HEAD`.
3. Make the working directory look like the index.

The downside of this approach is that it’s rewriting history, which can be problematic with a shared repository. Check out “[Los Peligros de Reorganizar](#)” for more on what can happen; the short version is that if other people have the commits you’re rewriting, you should probably avoid `reset`. This approach also won’t work if any other commits have been created since the merge; moving the refs would effectively lose those changes.

REVERSE THE COMMIT

If moving the branch pointers around isn’t going to work for you, Git gives you the option of making a new commit which undoes all the changes from an existing one. Git calls this operation a “revert”, and in this particular scenario, you’d invoke it like this:

```
$ git revert -m 1 HEAD
[master b1d8379] Revert "Merge branch 'topic'"
```

The `-m 1` flag indicates which parent is the “mainline” and should be kept. When you invoke a merge into HEAD (`git merge topic`), the new commit has two parents: the first one is HEAD (C6), and the second is the tip of the branch being merged in (C4). In this case, we want to undo all the changes introduced by merging in parent #2 (C4), while keeping all the content from parent #1 (C6).

The history with the revert commit looks like this:

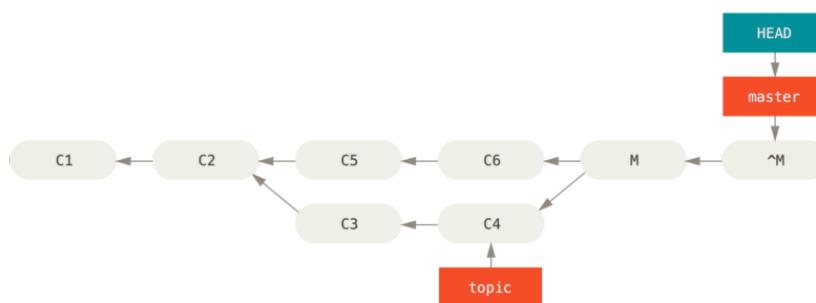


FIGURE 7-22
History after `git revert -m 1`

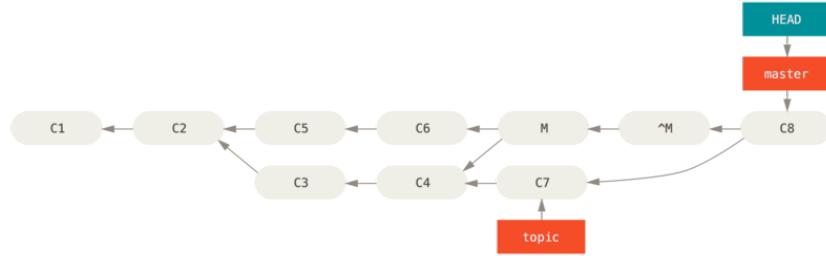
The new commit `^M` has exactly the same contents as `C6`, so starting from here it's as if the merge never happened, except that the now-unmerged commits are still in `HEAD`'s history. Git will get confused if you try to merge `topic` into `master` again:

```
$ git merge topic
Already up-to-date.
```

There's nothing in `topic` that isn't already reachable from `master`. What's worse, if you add work to `topic` and merge again, Git will only bring in the changes *since* the reverted merge:

FIGURE 7-23

History with a bad merge

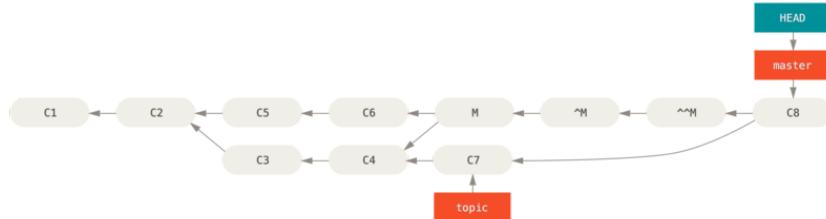


The best way around this is to un-revert the original merge, since now you want to bring in the changes that were reverted out, **then** create a new merge commit:

```
$ git revert ^M
[master 09f0126] Revert "Revert "Merge branch 'topic'''"
$ git merge topic
```

FIGURE 7-24

History after re-merging a reverted merge



In this example, M and ^M cancel out. ^^M effectively merges in the changes from C3 and C4, and C8 merges in the changes from C7, so now topic is fully merged.

Other Types of Merges

So far we've covered the normal merge of two branches, normally handled with what is called the “recursive” strategy of merging. There are other ways to merge branches together however. Let's cover a few of them quickly.

OUR OR THEIRS PREFERENCE

First of all, there is another useful thing we can do with the normal “recursive” mode of merging. We’ve already seen the `ignore-all-space` and `ignore-space-change` options which are passed with a `-X` but we can also tell Git to favor one side or the other when it sees a conflict.

By default, when Git sees a conflict between two branches being merged, it will add merge conflict markers into your code and mark the file as conflicted and let you resolve it. If you would prefer for Git to simply choose a specific side and ignore the other side instead of letting you manually merge the conflict, you can pass the `merge` command either a `-Xours` or `-Xtheirs`.

If Git sees this, it will not add conflict markers. Any differences that are mergeable, it will merge. Any differences that conflict, it will simply choose the side you specify in whole, including binary files.

If we go back to the “hello world” example we were using before, we can see that merging in our branch causes conflicts.

```
$ git merge mundo
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Resolved 'hello.rb' using previous resolution.
Automatic merge failed; fix conflicts and then commit the result.
```

However if we run it with `-Xours` or `-Xtheirs` it does not.

```
$ git merge -Xours mundo
Auto-merging hello.rb
Merge made by the 'recursive' strategy.
 hello.rb | 2 ++
 test.sh  | 2 ++
 2 files changed, 3 insertions(+), 1 deletion(-)
 create mode 100644 test.sh
```

In that case, instead of getting conflict markers in the file with “hello mundo” on one side and “hola world” on the other, it will simply pick “hola world”. However, all the other non-conflicting changes on that branch are merged successfully in.

This option can also be passed to the `git merge-file` command we saw earlier by running something like `git merge-file --ours` for individual file merges.

If you want to do something like this but not have Git even try to merge changes from the other side in, there is a more draconian option, which is the “ours” merge strategy. This is different from the “ours” recursive merge option.

This will basically do a fake merge. It will record a new merge commit with both branches as parents, but it will not even look at the branch you’re merging in. It will simply record as the result of the merge the exact code in your current branch.

```
$ git merge -s ours mundo
Merge made by the 'ours' strategy.
$ git diff HEAD HEAD~
$
```

You can see that there is no difference between the branch we were on and the result of the merge.

This can often be useful to basically trick Git into thinking that a branch is already merged when doing a merge later on. For example, say you branched off a “release” branch and have done some work on it that you will want to merge back into your “master” branch at some point. In the meantime some bugfix on “master” needs to be backported into your release branch. You can merge the bugfix branch into the `release` branch and also `merge -s ours` the same branch into your `master` branch (even though the fix is already there) so when you later merge the `release` branch again, there are no conflicts from the bugfix.

SUBTREE MERGING

The idea of the subtree merge is that you have two projects, and one of the projects maps to a subdirectory of the other one and vice versa. When you specify a subtree merge, Git is often smart enough to figure out that one is a subtree of the other and merge appropriately.

We’ll go through an example of adding a separate project into an existing project and then merging the code of the second into a subdirectory of the first.

First, we’ll add the Rack application to our project. We’ll add the Rack project as a remote reference in our own project and then check it out into its own branch:

```
$ git remote add rack_remote https://github.com/rack/rack
$ git fetch rack_remote
warning: no common commits
remote: Counting objects: 3184, done.
remote: Compressing objects: 100% (1465/1465), done.
```

```

remote: Total 3184 (delta 1952), reused 2770 (delta 1675)
Receiving objects: 100% (3184/3184), 677.42 KiB | 4 KiB/s, done.
Resolving deltas: 100% (1952/1952), done.
From https://github.com/rack/rack
 * [new branch]      build      -> rack_remote/build
 * [new branch]      master     -> rack_remote/master
 * [new branch]      rack-0.4   -> rack_remote/rack-0.4
 * [new branch]      rack-0.9   -> rack_remote/rack-0.9
$ git checkout -b rack_branch rack_remote/master
Branch rack_branch set up to track remote branch refs/remotes/rack_remote/master.
Switched to a new branch "rack_branch"

```

Now we have the root of the Rack project in our `rack_branch` branch and our own project in the `master` branch. If you check out one and then the other, you can see that they have different project roots:

```

$ ls
AUTHORS      KNOWN-ISSUES    Rakefile      contrib      lib
COPYING      README          bin           example     test
$ git checkout master
Switched to branch "master"
$ ls
README

```

This is sort of a strange concept. Not all the branches in your repository actually have to be branches of the same project. It's not common, because it's rarely helpful, but it's fairly easy to have branches contain completely different histories.

In this case, we want to pull the Rack project into our `master` project as a subdirectory. We can do that in Git with `git read-tree`. You'll learn more about `read-tree` and its friends in [Chapter 10](#), but for now know that it reads the root tree of one branch into your current staging area and working directory. We just switched back to your `master` branch, and we pull the `rack_branch` branch into the `rack` subdirectory of our `master` branch of our main project:

```
$ git read-tree --prefix=rack/ -u rack_branch
```

When we commit, it looks like we have all the Rack files under that subdirectory – as though we copied them in from a tarball. What gets interesting is that we can fairly easily merge changes from one of the branches to the other. So, if the Rack project updates, we can pull in upstream changes by switching to that branch and pulling:

```
$ git checkout rack_branch
$ git pull
```

Then, we can merge those changes back into our `master` branch. To pull in the changes and prepopulate the commit message, use the `--squash` and `--no-commit` options, as well as the recursive merge strategy's `-Xsubtree` option. (The recursive strategy is the default here, but we include it for clarity.)

```
$ git checkout master
$ git merge --squash -s recursive -Xsubtree=rack --no-commit rack_branch
Squash commit -- not updating HEAD
Automatic merge went well; stopped before committing as requested
```

All the changes from the Rack project are merged in and ready to be committed locally. You can also do the opposite – make changes in the `rack` subdirectory of your `master` branch and then merge them into your `rack_branch` branch later to submit them to the maintainers or push them upstream.

This gives us a way to have a workflow somewhat similar to the submodule workflow without using submodules (which we will cover in “[Submodules](#)”). We can keep branches with other related projects in our repository and subtree merge them into our project occasionally. It is nice in some ways, for example all the code is committed to a single place. However, it has other drawbacks in that it’s a bit more complex and easier to make mistakes in reintegrating changes or accidentally pushing a branch into an unrelated repository.

Another slightly weird thing is that to get a diff between what you have in your `rack` subdirectory and the code in your `rack_branch` branch – to see if you need to merge them – you can’t use the normal `diff` command. Instead, you must run `git diff-tree` with the branch you want to compare to:

```
$ git diff-tree -p rack_branch
```

Or, to compare what is in your `rack` subdirectory with what the `master` branch on the server was the last time you fetched, you can run

```
$ git diff-tree -p rack_remote/master
```

Rerere

The `git rerere` functionality is a bit of a hidden feature. The name stands for “reuse recorded resolution” and as the name implies, it allows you to ask Git to remember how you’ve resolved a hunk conflict so that the next time it sees the same conflict, Git can automatically resolve it for you.

There are a number of scenarios in which this functionality might be really handy. One of the examples that is mentioned in the documentation is if you want to make sure a long lived topic branch will merge cleanly but don’t want to have a bunch of intermediate merge commits. With `rerere` turned on you can merge occasionally, resolve the conflicts, then back out the merge. If you do this continuously, then the final merge should be easy because `rerere` can just do everything for you automatically.

This same tactic can be used if you want to keep a branch rebased so you don’t have to deal with the same rebasing conflicts each time you do it. Or if you want to take a branch that you merged and fixed a bunch of conflicts and then decide to rebase it instead - you likely won’t have to do all the same conflicts again.

Another situation is where you merge a bunch of evolving topic branches together into a testable head occasionally, as the Git project itself often does. If the tests fail, you can rewind the merges and re-do them without the topic branch that made the tests fail without having to re-resolve the conflicts again.

To enable the `rerere` functionality, you simply have to run this config setting:

```
$ git config --global rerere.enabled true
```

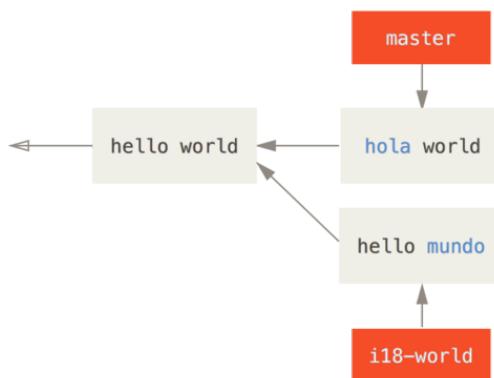
You can also turn it on by creating the `.git/rr-cache` directory in a specific repository, but the config setting is clearer and it can be done globally.

Now let’s see a simple example, similar to our previous one. Let’s say we have a file that looks like this:

```
#!/usr/bin/env ruby

def hello
  puts 'hello world'
end
```

In one branch we change the word “hello” to “hola”, then in another branch we change the “world” to “mundo”, just like before.

FIGURE 7-25

When we merge the two branches together, we'll get a merge conflict:

```
$ git merge i18n-world
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Recorded preimage for 'hello.rb'
Automatic merge failed; fix conflicts and then commit the result.
```

You should notice the new line `Recorded preimage for FILE` in there. Otherwise it should look exactly like a normal merge conflict. At this point, `rerere` can tell us a few things. Normally, you might run `git status` at this point to see what all conflicted:

```
$ git status
# On branch master
# Unmerged paths:
#   (use "git reset HEAD <file>..." to unstage)
#   (use "git add <file>..." to mark resolution)
#
#       both modified:    hello.rb
```

However, `git rerere` will also tell you what it has recorded the pre-merge state for with `git rerere status`:

```
$ git rerere status
hello.rb
```

And `git rerere diff` will show the current state of the resolution - what you started with to resolve and what you've resolved it to.

```
$ git rerere diff
--- a/hello.rb
+++ b/hello.rb
@@ -1,11 +1,11 @@
 #! /usr/bin/env ruby

 def hello
-<<<<<
- puts 'hello mundo'
-=====
+<<<<< HEAD
    puts 'hola mundo'
->>>>
+=====
+ puts 'hello mundo'
+>>>>> i18n-world
    end
```

Also (and this isn't really related to `rerere`), you can use `ls-files -u` to see the conflicted files and the before, left and right versions:

```
$ git ls-files -u
100644 39804c942a9c1f2c03dc7c5ebcd7f3e3a6b97519 1      hello.rb
100644 a440db6e8d1fd76ad438a49025a9ad9ce746f581 2      hello.rb
100644 54336ba847c3758ab604876419607e9443848474 3      hello.rb
```

Now you can resolve it to just be `puts 'hola mundo'` and you can run the `rerere diff` command again to see what `rerere` will remember:

```
$ git rerere diff
--- a/hello.rb
+++ b/hello.rb
@@ -1,11 +1,7 @@
#! /usr/bin/env ruby

def hello
-<<<<<
```

```

- puts 'hello mundo'
=====
- puts 'hola world'
->>>>>
+ puts 'hola mundo'
  end

```

So that basically says, when Git sees a hunk conflict in a `hello.rb` file that has “hello mundo” on one side and “hola world” on the other, it will resolve it to “hola mundo”.

Now we can mark it as resolved and commit it:

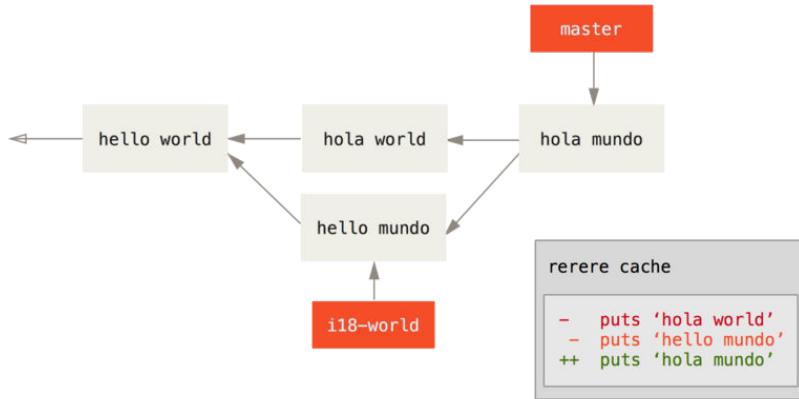
```

$ git add hello.rb
$ git commit
Recorded resolution for 'hello.rb'.
[master 68e16e5] Merge branch 'i18n'

```

You can see that it “Recorded resolution for FILE”.

FIGURE 7-26



Now, let’s undo that merge and then rebase it on top of our `master` branch instead. We can move our branch back by using `reset` as we saw in “[Reset Demystified](#)”.

```
$ git reset --hard HEAD^
HEAD is now at ad63f15 i18n the hello
```

Our merge is undone. Now let's rebase the topic branch.

```
$ git checkout i18n-world
Switched to branch 'i18n-world'

$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: i18n one word
Using index info to reconstruct a base tree...
Falling back to patching base and 3-way merge...
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Resolved 'hello.rb' using previous resolution.
Failed to merge in the changes.
Patch failed at 0001 i18n one word
```

Now, we got the same merge conflict like we expected, but take a look at the **Resolved FILE using previous resolution** line. If we look at the file, we'll see that it's already been resolved, there are no merge conflict markers in it.

```
$ cat hello.rb
#!/usr/bin/env ruby

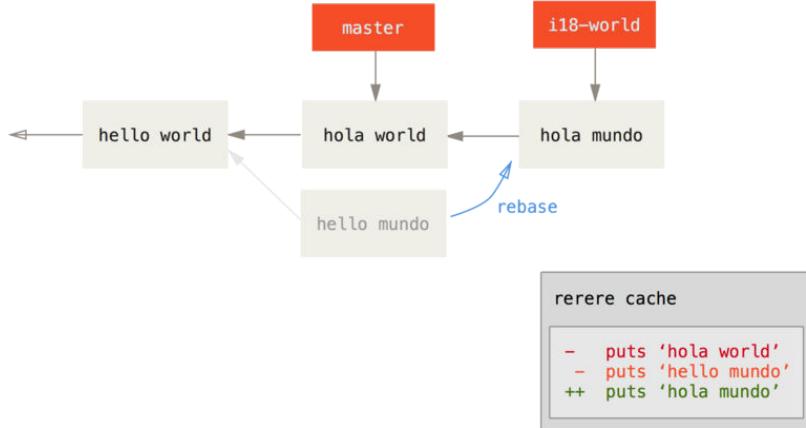
def hello
    puts 'hola mundo'
end
```

Also, **git diff** will show you how it was automatically re-resolved:

```
$ git diff
diff --cc hello.rb
index a440db6,54336ba..0000000
--- a/hello.rb
+++ b/hello.rb
@@@ -1,7 -1,7 +1,7 @@@
#! /usr/bin/env ruby

def hello
-   puts 'hola world'
```

```
- puts 'hello mundo'
++ puts 'hola mundo'
end
```

FIGURE 7-27

You can also recreate the conflicted file state with the `checkout` command:

```
$ git checkout --conflict=merge hello.rb
$ cat hello.rb
#!/usr/bin/env ruby

def hello
<<<<< ours
    puts 'hola mundo'
=====
    puts 'hello mundo'
>>>>> theirs
end
```

We saw an example of this in “Advanced Merging”. For now though, let’s resolve it by just running `rerere` again:

```
$ git rerere
Resolved 'hello.rb' using previous resolution.
$ cat hello.rb
#!/usr/bin/env ruby
```

```
def hello
  puts 'hola mundo'
end
```

We have re-resolved the file automatically using the `rerere` cached resolution. You can now add and continue the rebase to complete it.

```
$ git add hello.rb
$ git rebase --continue
Applying: i18n one word
```

So, if you do a lot of re-merges, or want to keep a topic branch up to date with your master branch without a ton of merges, or you rebase often, you can turn on `rerere` to help your life out a bit.

Debugging with Git

Git also provides a couple of tools to help you debug issues in your projects. Because Git is designed to work with nearly any type of project, these tools are pretty generic, but they can often help you hunt for a bug or culprit when things go wrong.

File Annotation

If you track down a bug in your code and want to know when it was introduced and why, file annotation is often your best tool. It shows you what commit was the last to modify each line of any file. So, if you see that a method in your code is buggy, you can annotate the file with `git blame` to see when each line of the method was last edited and by whom. This example uses the `-L` option to limit the output to lines 12 through 22:

```
$ git blame -L 12,22 simplegit.rb
^4832fe2 (Scott Chacon 2008-03-15 10:31:28 -0700 12) def show(tree = 'master')
^4832fe2 (Scott Chacon 2008-03-15 10:31:28 -0700 13)   command("git show #{tree}")
^4832fe2 (Scott Chacon 2008-03-15 10:31:28 -0700 14) end
^4832fe2 (Scott Chacon 2008-03-15 10:31:28 -0700 15)
9f6560e4 (Scott Chacon 2008-03-17 21:52:20 -0700 16) def log(tree = 'master')
79eaf55d (Scott Chacon 2008-04-06 10:15:08 -0700 17)   command("git log #{tree}")
9f6560e4 (Scott Chacon 2008-03-17 21:52:20 -0700 18) end
9f6560e4 (Scott Chacon 2008-03-17 21:52:20 -0700 19)
42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 20) def blame(path)
```

```
42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 21) command("git blame #{path}
42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 22) end
```

Notice that the first field is the partial SHA-1 of the commit that last modified that line. The next two fields are values extracted from that commit—the author name and the authored date of that commit—so you can easily see who modified that line and when. After that come the line number and the content of the file. Also note the ^4832fe2 commit lines, which designate that those lines were in this file’s original commit. That commit is when this file was first added to this project, and those lines have been unchanged since. This is a tad confusing, because now you’ve seen at least three different ways that Git uses the ^ to modify a commit SHA-1, but that is what it means here.

Another cool thing about Git is that it doesn’t track file renames explicitly. It records the snapshots and then tries to figure out what was renamed implicitly, after the fact. One of the interesting features of this is that you can ask it to figure out all sorts of code movement as well. If you pass -C to `git blame`, Git analyzes the file you’re annotating and tries to figure out where snippets of code within it originally came from if they were copied from elsewhere. For example, say you are refactoring a file named `GITServerHandler.m` into multiple files, one of which is `GITPackUpload.m`. By blaming `GITPackUpload.m` with the -C option, you can see where sections of the code originally came from:

```
$ git blame -C -L 141,153 GITPackUpload.m
f344f58d GITServerHandler.m (Scott 2009-01-04 141)
f344f58d GITServerHandler.m (Scott 2009-01-04 142) - (void) gatherObjectShasFromC
f344f58d GITServerHandler.m (Scott 2009-01-04 143) {
70befddd GITServerHandler.m (Scott 2009-03-22 144) //NSLog(@"%@", GATHER COMMIT)
ad11ac80 GITPackUpload.m (Scott 2009-03-24 145)
ad11ac80 GITPackUpload.m (Scott 2009-03-24 146) NSString *parentSha;
ad11ac80 GITPackUpload.m (Scott 2009-03-24 147) GITCommit *commit = [g
ad11ac80 GITPackUpload.m (Scott 2009-03-24 148)
ad11ac80 GITPackUpload.m (Scott 2009-03-24 149) //NSLog(@"%@", GATHER COMMIT)
ad11ac80 GITPackUpload.m (Scott 2009-03-24 150)
56ef2caf GITServerHandler.m (Scott 2009-01-05 151) if(commit) {
56ef2caf GITServerHandler.m (Scott 2009-01-05 152) [refDict setOb
56ef2caf GITServerHandler.m (Scott 2009-01-05 153)
```

This is really useful. Normally, you get as the original commit the commit where you copied the code over, because that is the first time you touched those lines in this file. Git tells you the original commit where you wrote those lines, even if it was in another file.

Binary Search

Annotating a file helps if you know where the issue is to begin with. If you don't know what is breaking, and there have been dozens or hundreds of commits since the last state where you know the code worked, you'll likely turn to `git bisect` for help. The `bisect` command does a binary search through your commit history to help you identify as quickly as possible which commit introduced an issue.

Let's say you just pushed out a release of your code to a production environment, you're getting bug reports about something that wasn't happening in your development environment, and you can't imagine why the code is doing that. You go back to your code, and it turns out you can reproduce the issue, but you can't figure out what is going wrong. You can bisect the code to find out. First you run `git bisect start` to get things going, and then you use `git bisect bad` to tell the system that the current commit you're on is broken. Then, you must tell bisect when the last known good state was, using `git bisect good [good_commit]`:

```
$ git bisect start
$ git bisect bad
$ git bisect good v1.0
Bisecting: 6 revisions left to test after this
[ecb6e1bc347ccecc5f9350d878ce677feb13d3b2] error handling on repo
```

Git figured out that about 12 commits came between the commit you marked as the last good commit (v1.0) and the current bad version, and it checked out the middle one for you. At this point, you can run your test to see if the issue exists as of this commit. If it does, then it was introduced sometime before this middle commit; if it doesn't, then the problem was introduced sometime after the middle commit. It turns out there is no issue here, and you tell Git that by typing `git bisect good` and continue your journey:

```
$ git bisect good
Bisecting: 3 revisions left to test after this
[b047b02ea83310a70fd603dc8cd7a6cd13d15c04] secure this thing
```

Now you're on another commit, halfway between the one you just tested and your bad commit. You run your test again and find that this commit is broken, so you tell Git that with `git bisect bad`:

```
$ git bisect bad
Bisecting: 1 revisions left to test after this
[f71ce38690acf49c1f3c9bea38e09d82a5ce6014] drop exceptions table
```

This commit is fine, and now Git has all the information it needs to determine where the issue was introduced. It tells you the SHA-1 of the first bad commit and show some of the commit information and which files were modified in that commit so you can figure out what happened that may have introduced this bug:

```
$ git bisect good
b047b02ea83310a70fd603dc8cd7a6cd13d15c04 is first bad commit
commit b047b02ea83310a70fd603dc8cd7a6cd13d15c04
Author: PJ Hyett <pjhyett@example.com>
Date:   Tue Jan 27 14:48:32 2009 -0800

    secure this thing

:040000 040000 40ee3e7821b895e52c1695092db9bdc4c61d1730
f24d3c6ebfc639b1a3814550e62d60b8e68a8e4 M config
```

When you're finished, you should run `git bisect reset` to reset your HEAD to where you were before you started, or you'll end up in a weird state:

```
$ git bisect reset
```

This is a powerful tool that can help you check hundreds of commits for an introduced bug in minutes. In fact, if you have a script that will exit 0 if the project is good or non-0 if the project is bad, you can fully automate `git bisect`. First, you again tell it the scope of the bisect by providing the known bad and good commits. You can do this by listing them with the `bisect start` command if you want, listing the known bad commit first and the known good commit second:

```
$ git bisect start HEAD v1.0
$ git bisect run test-error.sh
```

Doing so automatically runs `test-error.sh` on each checked-out commit until Git finds the first broken commit. You can also run something like `make` or `make tests` or whatever you have that runs automated tests for you.

Submodules

It often happens that while working on one project, you need to use another project from within it. Perhaps it's a library that a third party developed or that you're developing separately and using in multiple parent projects. A common issue arises in these scenarios: you want to be able to treat the two projects as separate yet still be able to use one from within the other.

Here's an example. Suppose you're developing a web site and creating Atom feeds. Instead of writing your own Atom-generating code, you decide to use a library. You're likely to have to either include this code from a shared library like a CPAN install or Ruby gem, or copy the source code into your own project tree. The issue with including the library is that it's difficult to customize the library in any way and often more difficult to deploy it, because you need to make sure every client has that library available. The issue with vendorizing the code into your own project is that any custom changes you make are difficult to merge when upstream changes become available.

Git addresses this issue using submodules. Submodules allow you to keep a Git repository as a subdirectory of another Git repository. This lets you clone another repository into your project and keep your commits separate.

Starting with Submodules

We'll walk through developing a simple project that has been split up into a main project and a few sub-projects.

Let's start by adding an existing Git repository as a submodule of the repository that we're working on. To add a new submodule you use the `git submodule add` command with the URL of the project you would like to start tracking. In this example, we'll add a library called "DbConnector".

```
$ git submodule add https://github.com/chaconinc/DbConnector
Cloning into 'DbConnector'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
Unpacking objects: 100% (11/11), done.
Checking connectivity... done.
```

By default, submodules will add the subproject into a directory named the same as the repository, in this case "DbConnector". You can add a different path at the end of the command if you want it to go elsewhere.

If you run `git status` at this point, you'll notice a few things.

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

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:  .gitmodules
    new file:  DbConnector
```

First you should notice the new `.gitmodules` file. This is a configuration file that stores the mapping between the project's URL and the local subdirectory you've pulled it into:

```
$ cat .gitmodules
[submodule "DbConnector"]
  path = DbConnector
  url = https://github.com/chaconinc/DbConnector
```

If you have multiple submodules, you'll have multiple entries in this file. It's important to note that this file is version-controlled with your other files, like your `.gitignore` file. It's pushed and pulled with the rest of your project. This is how other people who clone this project know where to get the submodule projects from.

Since the URL in the `.gitmodules` file is what other people will first try to clone/fetch from, make sure to use a URL that they can access if possible. For example, if you use a different URL to push to than others would to pull from, use the one that others have access to. You can overwrite this value locally with `git config submodule.DbConnector.url PRIVATE_URL` for your own use.

The other listing in the `git status` output is the project folder entry. If you run `git diff` on that, you see something interesting:

```
$ git diff --cached DbConnector
diff --git a/DbConnector b/DbConnector
new file mode 160000
index 000000..c3f01dc
--- /dev/null
+++ b/DbConnector
```

```
@@ -0,0 +1 @@
+Subproject commit c3f01dc8862123d317dd46284b05b6892c7b29bc
```

Although `DbConnector` is a subdirectory in your working directory, Git sees it as a submodule and doesn't track its contents when you're not in that directory. Instead, Git sees it as a particular commit from that repository.

If you want a little nicer diff output, you can pass the `--submodule` option to `git diff`.

```
$ git diff --cached --submodule
diff --git a/.gitmodules b/.gitmodules
new file mode 100644
index 0000000..71fc376
--- /dev/null
+++ b/.gitmodules
@@ -0,0 +1,3 @@
+[submodule "DbConnector"]
+    path = DbConnector
+    url = https://github.com/chaconinc/DbConnector
Submodule DbConnector 0000000...c3f01dc (new submodule)
```

When you commit, you see something like this:

```
$ git commit -am 'added DbConnector module'
[master fb9093c] added DbConnector module
2 files changed, 4 insertions(+)
create mode 100644 .gitmodules
create mode 160000 DbConnector
```

Notice the `160000` mode for the `DbConnector` entry. That is a special mode in Git that basically means you're recording a commit as a directory entry rather than a subdirectory or a file.

Cloning a Project with Submodules

Here we'll clone a project with a submodule in it. When you clone such a project, by default you get the directories that contain submodules, but none of the files within them yet:

```
$ git clone https://github.com/chaconinc/MainProject
Cloning into 'MainProject'...
remote: Counting objects: 14, done.
```

```

remote: Compressing objects: 100% (13/13), done.
remote: Total 14 (delta 1), reused 13 (delta 0)
Unpacking objects: 100% (14/14), done.
Checking connectivity... done.
$ cd MainProject
$ ls -la
total 16
drwxr-xr-x  9 schacon  staff  306 Sep 17 15:21 .
drwxr-xr-x  7 schacon  staff  238 Sep 17 15:21 ..
drwxr-xr-x 13 schacon  staff  442 Sep 17 15:21 .git
-rw-r--r--  1 schacon  staff   92 Sep 17 15:21 .gitmodules
drwxr-xr-x  2 schacon  staff   68 Sep 17 15:21 DbConnector
-rw-r--r--  1 schacon  staff  756 Sep 17 15:21 Makefile
drwxr-xr-x  3 schacon  staff  102 Sep 17 15:21 includes
drwxr-xr-x  4 schacon  staff  136 Sep 17 15:21 scripts
drwxr-xr-x  4 schacon  staff  136 Sep 17 15:21 src
$ cd DbConnector/
$ ls
$
```

The `DbConnector` directory is there, but empty. You must run two commands: `git submodule init` to initialize your local configuration file, and `git submodule update` to fetch all the data from that project and check out the appropriate commit listed in your superproject:

```

$ git submodule init
Submodule 'DbConnector' (https://github.com/chaconinc/DbConnector) registered for
$ git submodule update
Cloning into 'DbConnector'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
Unpacking objects: 100% (11/11), done.
Checking connectivity... done.
Submodule path 'DbConnector': checked out 'c3f01dc8862123d317dd46284b05b6892c7b29b'
```

Now your `DbConnector` subdirectory is at the exact state it was in when you committed earlier.

There is another way to do this which is a little simpler, however. If you pass `--recursive` to the `git clone` command, it will automatically initialize and update each submodule in the repository.

```

$ git clone --recursive https://github.com/chaconinc/MainProject
Cloning into 'MainProject'...
remote: Counting objects: 14, done.
```

```

remote: Compressing objects: 100% (13/13), done.
remote: Total 14 (delta 1), reused 13 (delta 0)
Unpacking objects: 100% (14/14), done.
Checking connectivity... done.
Submodule 'DbConnector' (https://github.com/chaconinc/DbConnector) registered for path 'DbConnector'
Cloning into 'DbConnector',...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
Unpacking objects: 100% (11/11), done.
Checking connectivity... done.
Submodule path 'DbConnector': checked out 'c3f01dc8862123d317dd46284b05b6892c7b29bc'
```

Working on a Project with Submodules

Now we have a copy of a project with submodules in it and will collaborate with our teammates on both the main project and the submodule project.

PULLING IN UPSTREAM CHANGES

The simplest model of using submodules in a project would be if you were simply consuming a subproject and wanted to get updates from it from time to time but were not actually modifying anything in your checkout. Let's walk through a simple example there.

If you want to check for new work in a submodule, you can go into the directory and run `git fetch` and `git merge` the upstream branch to update the local code.

```

$ git fetch
From https://github.com/chaconinc/DbConnector
  c3f01dc..d0354fc  master      -> origin/master
$ git merge origin/master
Updating c3f01dc..d0354fc
Fast-forward
  scripts/connect.sh | 1 +
  src/db.c          | 1 +
  2 files changed, 2 insertions(+)
```

Now if you go back into the main project and run `git diff --submodule` you can see that the submodule was updated and get a list of commits that were added to it. If you don't want to type `--submodule` every time you run `git diff`, you can set it as the default format by setting the `diff.submodule` config value to "log".

```
$ git config --global diff.submodule log
$ git diff
Submodule DbConnector c3f01dc..d0354fc:
  > more efficient db routine
  > better connection routine
```

If you commit at this point then you will lock the submodule into having the new code when other people update.

There is an easier way to do this as well, if you prefer to not manually fetch and merge in the subdirectory. If you run `git submodule update --remote`, Git will go into your submodules and fetch and update for you.

```
$ git submodule update --remote DbConnector
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 4 (delta 2), reused 4 (delta 2)
Unpacking objects: 100% (4/4), done.
From https://github.com/chaconinc/DbConnector
  3f19983..d0354fc master      -> origin/master
Submodule path 'DbConnector': checked out 'd0354fc054692d3906c85c3af05ddce39a1c064'
```

This command will by default assume that you want to update the checkout to the `master` branch of the submodule repository. You can, however, set this to something different if you want. For example, if you want to have the `DbConnector` submodule track that repository's "stable" branch, you can set it in either your `.gitmodules` file (so everyone else also tracks it), or just in your local `.git/config` file. Let's set it in the `.gitmodules` file:

```
$ git config -f .gitmodules submodule.DbConnector.branch stable
$ git submodule update --remote
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 4 (delta 2), reused 4 (delta 2)
Unpacking objects: 100% (4/4), done.
From https://github.com/chaconinc/DbConnector
  27cf5d3..c87d55d stable -> origin/stable
Submodule path 'DbConnector': checked out 'c87d55d4c6d4b05ee34fbc8cb6f7bf4585ae668'
```

If you leave off the `-f .gitmodules` it will only make the change for you, but it probably makes more sense to track that information with the repository so everyone else does as well.

When we run `git status` at this point, Git will show us that we have “new commits” on the submodule.

```
$ 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:   .gitmodules
    modified:   DbConnector (new commits)

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

If you set the configuration setting `status.submodulessummary`, Git will also show you a short summary of changes to your submodules:

```
$ git config status.submodulessummary 1

$ 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:   .gitmodules
    modified:   DbConnector (new commits)

Submodules changed but not updated:

* DbConnector c3f01dc...c87d55d (4):
  > catch non-null terminated lines
```

At this point if you run `git diff` we can see both that we have modified our `.gitmodules` file and also that there are a number of commits that we’ve pulled down and are ready to commit to our submodule project.

```
$ git diff
diff --git a/.gitmodules b/.gitmodules
index 6fc0b3d..fd1cc29 100644
```

```

--- a/.gitmodules
+++ b/.gitmodules
@@ -1,3 +1,4 @@
[submodule "DbConnector"]
    path = DbConnector
    url = https://github.com/chaconinc/DbConnector
+   branch = stable
Submodule DbConnector c3f01dc..c87d55d:
    > catch non-null terminated lines
    > more robust error handling
    > more efficient db routine
    > better connection routine

```

This is pretty cool as we can actually see the log of commits that we're about to commit to in our submodule. Once committed, you can see this information after the fact as well when you run `git log -p`.

```

$ git log -p --submodule
commit 0a24cfcc121a8a3c118e0105ae4ae4c00281cf7ae
Author: Scott Chacon <schacon@gmail.com>
Date:   Wed Sep 17 16:37:02 2014 +0200

        updating DbConnector for bug fixes

diff --git a/.gitmodules b/.gitmodules
index 6fc0b3d..fd1cc29 100644
--- a/.gitmodules
+++ b/.gitmodules
@@ -1,3 +1,4 @@
[submodule "DbConnector"]
    path = DbConnector
    url = https://github.com/chaconinc/DbConnector
+   branch = stable
Submodule DbConnector c3f01dc..c87d55d:
    > catch non-null terminated lines
    > more robust error handling
    > more efficient db routine
    > better connection routine

```

Git will by default try to update **all** of your submodules when you run `git submodule update --remote` so if you have a lot of them, you may want to pass the name of just the submodule you want to try to update.

WORKING ON A SUBMODULE

It's quite likely that if you're using submodules, you're doing so because you really want to work on the code in the submodule at the same time as you're working on the code in the main project (or across several submodules). Otherwise you would probably instead be using a simpler dependency management system (such as Maven or Rubygems).

So now let's go through an example of making changes to the submodule at the same time as the main project and committing and publishing those changes at the same time.

So far, when we've run the `git submodule update` command to fetch changes from the submodule repositories, Git would get the changes and update the files in the subdirectory but will leave the sub-repository in what's called a "detached HEAD" state. This means that there is no local working branch (like "master", for example) tracking changes. So any changes you make aren't being tracked well.

In order to set up your submodule to be easier to go in and hack on, you need do two things. You need to go into each submodule and check out a branch to work on. Then you need to tell Git what to do if you have made changes and then `git submodule update --remote` pulls in new work from upstream. The options are that you can merge them into your local work, or you can try to rebase your local work on top of the new changes.

First of all, let's go into our submodule directory and check out a branch.

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

Let's try it with the "merge" option. To specify it manually, we can just add the `--merge` option to our `update` call. Here we'll see that there was a change on the server for this submodule and it gets merged in.

```
$ git submodule update --remote --merge  
remote: Counting objects: 4, done.  
remote: Compressing objects: 100% (2/2), done.  
remote: Total 4 (delta 2), reused 4 (delta 2)  
Unpacking objects: 100% (4/4), done.  
From https://github.com/chaconinc/DbConnector  
  c87d55d..92c7337  stable    -> origin/stable  
Updating c87d55d..92c7337  
Fast-forward  
 src/main.c | 1 +
```

```
1 file changed, 1 insertion(+)
Submodule path 'DbConnector': merged in '92c7337b30ef9e0893e758dac2459d07362ab5ea'
```

If we go into the DbConnector directory, we have the new changes already merged into our local `stable` branch. Now let's see what happens when we make our own local change to the library and someone else pushes another change upstream at the same time.

```
$ cd DbConnector/
$ vim src/db.c
$ git commit -am 'unicode support'
[stable f906e16] unicode support
 1 file changed, 1 insertion(+)
```

Now if we update our submodule we can see what happens when we have made a local change and upstream also has a change we need to incorporate.

```
$ git submodule update --remote --rebase
First, rewinding head to replay your work on top of it...
Applying: unicode support
Submodule path 'DbConnector': rebased into '5d60ef9bbebf5a0c1c1050f242ceeb54ad58da'
```

If you forget the `--rebase` or `--merge`, Git will just update the submodule to whatever is on the server and reset your project to a detached HEAD state.

```
$ git submodule update --remote
Submodule path 'DbConnector': checked out '5d60ef9bbebf5a0c1c1050f242ceeb54ad58da'
```

If this happens, don't worry, you can simply go back into the directory and check out your branch again (which will still contain your work) and merge or rebase `origin/stable` (or whatever remote branch you want) manually.

If you haven't committed your changes in your submodule and you run a submodule update that would cause issues, Git will fetch the changes but not overwrite unsaved work in your submodule directory.

```
$ git submodule update --remote
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 4 (delta 0), reused 4 (delta 0)
Unpacking objects: 100% (4/4), done.
From https://github.com/chaconinc/DbConnector
```

```
5d60ef9..c75e92a stable -> origin/stable
error: Your local changes to the following files would be overwritten by checkout:
  scripts/setup.sh
Please, commit your changes or stash them before you can switch branches.
Aborting
Unable to checkout 'c75e92a2b3855c9e5b66f915308390d9db204aca' in submodule path 'DbConnector'
```

If you made changes that conflict with something changed upstream, Git will let you know when you run the update.

```
$ git submodule update --remote --merge
Auto-merging scripts/setup.sh
CONFLICT (content): Merge conflict in scripts/setup.sh
Recorded preimage for 'scripts/setup.sh'
Automatic merge failed; fix conflicts and then commit the result.
Unable to merge 'c75e92a2b3855c9e5b66f915308390d9db204aca' in submodule path 'DbConnector'
```

You can go into the submodule directory and fix the conflict just as you normally would.

PUBLISHING SUBMODULE CHANGES

Now we have some changes in our submodule directory. Some of these were brought in from upstream by our updates and others were made locally and aren't available to anyone else yet as we haven't pushed them yet.

```
$ git diff
Submodule DbConnector c87d55d..82d2ad3:
> Merge from origin/stable
> updated setup script
> unicode support
> remove unnecessary method
> add new option for conn pooling
```

If we commit in the main project and push it up without pushing the submodule changes up as well, other people who try to check out our changes are going to be in trouble since they will have no way to get the submodule changes that are depended on. Those changes will only exist on our local copy.

In order to make sure this doesn't happen, you can ask Git to check that all your submodules have been pushed properly before pushing the main project. The `git push` command takes the `--recurse-submodules` argument which can be set to either "check" or "on-demand". The "check" option will make

push simply fail if any of the committed submodule changes haven't been pushed.

```
$ git push --recurse-submodules=check
The following submodule paths contain changes that can
not be found on any remote:
  DbConnector

Please try

    git push --recurse-submodules=on-demand

or cd to the path and use

    git push

to push them to a remote.
```

As you can see, it also gives us some helpful advice on what we might want to do next. The simple option is to go into each submodule and manually push to the remotes to make sure they're externally available and then try this push again.

The other option is to use the “on-demand” value, which will try to do this for you.

```
$ git push --recurse-submodules=on-demand
Pushing submodule 'DbConnector'
Counting objects: 9, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (8/8), done.
Writing objects: 100% (9/9), 917 bytes | 0 bytes/s, done.
Total 9 (delta 3), reused 0 (delta 0)
To https://github.com/chaconinc/DbConnector
  c75e92a..82d2ad3  stable -> stable
Counting objects: 2, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (2/2), 266 bytes | 0 bytes/s, done.
Total 2 (delta 1), reused 0 (delta 0)
To https://github.com/chaconinc/MainProject
  3d6d338..9a377d1  master -> master
```

As you can see there, Git went into the DbConnector module and pushed it before pushing the main project. If that submodule push fails for some reason, the main project push will also fail.

MERGING SUBMODULE CHANGES

If you change a submodule reference at the same time as someone else, you may run into some problems. That is, if the submodule histories have diverged and are committed to diverging branches in a superproject, it may take a bit of work for you to fix.

If one of the commits is a direct ancestor of the other (a fast-forward merge), then Git will simply choose the latter for the merge, so that works fine.

Git will not attempt even a trivial merge for you, however. If the submodule commits diverge and need to be merged, you will get something that looks like this:

```
$ git pull
remote: Counting objects: 2, done.
remote: Compressing objects: 100% (1/1), done.
remote: Total 2 (delta 1), reused 2 (delta 1)
Unpacking objects: 100% (2/2), done.
From https://github.com/chaconinc/MainProject
  9a377d1..eb974f8  master      -> origin/master
Fetching submodule DbConnector
warning: Failed to merge submodule DbConnector (merge following commits not found)
Auto-merging DbConnector
CONFLICT (submodule): Merge conflict in DbConnector
Automatic merge failed; fix conflicts and then commit the result.
```

So basically what has happened here is that Git has figured out that the two branches record points in the submodule's history that are divergent and need to be merged. It explains it as "merge following commits not found", which is confusing but we'll explain why that is in a bit.

To solve the problem, you need to figure out what state the submodule should be in. Strangely, Git doesn't really give you much information to help out here, not even the SHA-1s of the commits of both sides of the history. Fortunately, it's simple to figure out. If you run `git diff` you can get the SHA-1s of the commits recorded in both branches you were trying to merge.

```
$ git diff
diff --cc DbConnector
index eb41d76,c771610..0000000
--- a/DbConnector
+++ b/DbConnector
```

So, in this case, `eb41d76` is the commit in our submodule that **we** had and `c771610` is the commit that upstream had. If we go into our submodule direc-

ry, it should already be on eb41d76 as the merge would not have touched it. If for whatever reason it's not, you can simply create and checkout a branch pointing to it.

What is important is the SHA-1 of the commit from the other side. This is what you'll have to merge in and resolve. You can either just try the merge with the SHA-1 directly, or you can create a branch for it and then try to merge that in. We would suggest the latter, even if only to make a nicer merge commit message.

So, we will go into our submodule directory, create a branch based on that second SHA-1 from `git diff` and manually merge.

```
$ cd DbConnector
$ git rev-parse HEAD
eb41d764bccf88be77aced643c13a7fa86714135

$ git branch try-merge c771610
(DbConnector) $ git merge try-merge
Auto-merging src/main.c
CONFLICT (content): Merge conflict in src/main.c
Recorded preimage for 'src/main.c'
Automatic merge failed; fix conflicts and then commit the result.
```

We got an actual merge conflict here, so if we resolve that and commit it, then we can simply update the main project with the result.

```
$ vim src/main.c ①
$ git add src/main.c
$ git commit -am 'merged our changes'
Recorded resolution for 'src/main.c'.
[master 9fd905e] merged our changes

$ cd .. ②
$ git diff ③
diff --cc DbConnector
index eb41d76,c771610..0000000
--- a/DbConnector
+++ b/DbConnector
@@@ -1,1 -1,1 +1,1 @@@
- Subproject commit eb41d764bccf88be77aced643c13a7fa86714135
- Subproject commit c77161012afbbe1f58b5053316ead08f4b7e6d1d
++Subproject commit 9fd905e5d7f45a0d4cbc43d1ee550f16a30e825a
$ git add DbConnector ④
```

```
$ git commit -m "Merge Tom's Changes" ⑤
[master 10d2c60] Merge Tom's Changes
```

- ① First we resolve the conflict
- ② Then we go back to the main project directory
- ③ We can check the SHA-1s again
- ④ Resolve the conflicted submodule entry
- ⑤ Commit our merge

It can be a bit confusing, but it's really not very hard.

Interestingly, there is another case that Git handles. If a merge commit exists in the submodule directory that contains **both** commits in its history, Git will suggest it to you as a possible solution. It sees that at some point in the submodule project, someone merged branches containing these two commits, so maybe you'll want that one.

This is why the error message from before was “merge following commits not found”, because it could not do **this**. It's confusing because who would expect it to **try to do this**?

If it does find a single acceptable merge commit, you'll see something like this:

```
$ git merge origin/master
warning: Failed to merge submodule DbConnector (not fast-forward)
Found a possible merge resolution for the submodule:
  9fd905e5d7f45a0d4cbc43d1ee550f16a30e825a: > merged our changes
If this is correct simply add it to the index for example
by using:

  git update-index --cacheinfo 160000 9fd905e5d7f45a0d4cbc43d1ee550f16a30e825a "DbConnector"
which will accept this suggestion.
Auto-merging DbConnector
CONFLICT (submodule): Merge conflict in DbConnector
Automatic merge failed; fix conflicts and then commit the result.
```

What it's suggesting that you do is to update the index like you had run `git add`, which clears the conflict, then commit. You probably shouldn't do this though. You can just as easily go into the submodule directory, see what the difference is, fast-forward to this commit, test it properly, and then commit it.

```
$ cd DbConnector/
$ git merge 9fd905e
Updating eb41d76..9fd905e
Fast-forward

$ cd ..
$ git add DbConnector
$ git commit -am 'Fast forwarded to a common submodule child'
```

This accomplishes the same thing, but at least this way you can verify that it works and you have the code in your submodule directory when you're done.

Submodule Tips

There are a few things you can do to make working with submodules a little easier.

SUBMODULE FOREACH

There is a `foreach` submodule command to run some arbitrary command in each submodule. This can be really helpful if you have a number of submodules in the same project.

For example, let's say we want to start a new feature or do a bugfix and we have work going on in several submodules. We can easily stash all the work in all our submodules.

```
$ git submodule foreach 'git stash'
Entering 'CryptoLibrary'
No local changes to save
Entering 'DbConnector'
Saved working directory and index state WIP on stable: 82d2ad3 Merge from origin/stable
HEAD is now at 82d2ad3 Merge from origin/stable
```

Then we can create a new branch and switch to it in all our submodules.

```
$ git submodule foreach 'git checkout -b featureA'
Entering 'CryptoLibrary'
Switched to a new branch 'featureA'
Entering 'DbConnector'
Switched to a new branch 'featureA'
```

You get the idea. One really useful thing you can do is produce a nice unified diff of what is changed in your main project and all your subprojects as well.

```
$ git diff; git submodule foreach 'git diff'
Submodule DbConnector contains modified content
diff --git a/src/main.c b/src/main.c
index 210f1ae..1f0acdc 100644
--- a/src/main.c
+++ b/src/main.c
@@ -245,6 +245,8 @@ static int handle_alias(int *argcp, const char ***argv)

    commit_pager_choice();

+    url = url_decode(url_orig);
+
+    /* build alias_argv */
+    alias_argv = xmalloc(sizeof(*alias_argv) * (argc + 1));
+    alias_argv[0] = alias_string + 1;
Entering 'DbConnector'
diff --git a/src/db.c b/src/db.c
index 1aaefb6..5297645 100644
--- a/src/db.c
+++ b/src/db.c
@@ -93,6 +93,11 @@ char *url_decode_mem(const char *url, int len)
    return url_decode_internal(&url, len, NULL, &out, 0);
}

+char *url_decode(const char *url)
+{
+    return url_decode_mem(url, strlen(url));
+}
+
char *url_decode_parameter_name(const char **query)
{
    struct strbuf out = STRBUF_INIT;
```

Here we can see that we're defining a function in a submodule and calling it in the main project. This is obviously a simplified example, but hopefully it gives you an idea of how this may be useful.

USEFUL ALIASES

You may want to set up some aliases for some of these commands as they can be quite long and you can't set configuration options for most of them to make them defaults. We covered setting up Git aliases in “[Alias de Git](#)”, but here is an

example of what you may want to set up if you plan on working with submodules in Git a lot.

```
$ git config alias.sdiff '!"git diff && git submodule foreach "git diff"'
$ git config alias.spush 'push --recurse-submodules=on-demand'
$ git config alias.supdate 'submodule update --remote --merge'
```

This way you can simply run `git supdate` when you want to update your submodules, or `git spush` to push with submodule dependency checking.

Issues with Submodules

Using submodules isn't without hiccups, however.

For instance switching branches with submodules in them can also be tricky. If you create a new branch, add a submodule there, and then switch back to a branch without that submodule, you still have the submodule directory as an untracked directory:

```
$ git checkout -b add-crypto
Switched to a new branch 'add-crypto'

$ git submodule add https://github.com/chaconinc/CryptoLibrary
Cloning into 'CryptoLibrary'...
...

$ git commit -am 'adding crypto library'
[add-crypto 4445836] adding crypto library
 2 files changed, 4 insertions(+)
 create mode 160000 CryptoLibrary

$ git checkout master
warning: unable to rmdir CryptoLibrary: Directory not empty
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.

$ 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)

    CryptoLibrary/
```

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

Removing the directory isn't difficult, but it can be a bit confusing to have that in there. If you do remove it and then switch back to the branch that has that submodule, you will need to run `submodule update --init` to repopulate it.

```
$ git clean -fdx
Removing CryptoLibrary/

$ git checkout add-crypto
Switched to branch 'add-crypto'

$ ls CryptoLibrary/
$ git submodule update --init
Submodule path 'CryptoLibrary': checked out 'b8dda6aa182ea4464f3f3264b11e0268545172af'

$ ls CryptoLibrary/
Makefile      includes      scripts      src
```

Again, not really very difficult, but it can be a little confusing.

The other main caveat that many people run into involves switching from subdirectories to submodules. If you've been tracking files in your project and you want to move them out into a submodule, you must be careful or Git will get angry at you. Assume that you have files in a subdirectory of your project, and you want to switch it to a submodule. If you delete the subdirectory and then run `submodule add`, Git yells at you:

```
$ rm -Rf CryptoLibrary/
$ git submodule add https://github.com/chaconinc/CryptoLibrary
'CryptoLibrary' already exists in the index
```

You have to unstage the `CryptoLibrary` directory first. Then you can add the submodule:

```
$ git rm -r CryptoLibrary
$ git submodule add https://github.com/chaconinc/CryptoLibrary
Cloning into 'CryptoLibrary'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
```

```
Unpacking objects: 100% (11/11), done.
Checking connectivity... done.
```

Now suppose you did that in a branch. If you try to switch back to a branch where those files are still in the actual tree rather than a submodule – you get this error:

```
$ git checkout master
error: The following untracked working tree files would be overwritten by checkout
  CryptoLibrary/Makefile
  CryptoLibrary/includes/crypto.h
...
Please move or remove them before you can switch branches.
Aborting
```

You can force it to switch with `checkout -f`, but be careful that you don't have unsaved changes in there as they could be overwritten with that command.

```
$ git checkout -f master
warning: unable to rmdir CryptoLibrary: Directory not empty
Switched to branch 'master'
```

Then, when you switch back, you get an empty `CryptoLibrary` directory for some reason and `git submodule update` may not fix it either. You may need to go into your submodule directory and run a `git checkout .` to get all your files back. You could run this in a `submodule foreach` script to run it for multiple submodules.

It's important to note that submodules these days keep all their Git data in the top project's `.git` directory, so unlike much older versions of Git, destroying a submodule directory won't lose any commits or branches that you had.

With these tools, submodules can be a fairly simple and effective method for developing on several related but still separate projects simultaneously.

Bundling

Though we've covered the common ways to transfer Git data over a network (HTTP, SSH, etc), there is actually one more way to do so that is not commonly used but can actually be quite useful.

Git is capable of “bundling” its data into a single file. This can be useful in various scenarios. Maybe your network is down and you want to send changes to your co-workers. Perhaps you’re working somewhere offsite and don’t have access to the local network for security reasons. Maybe your wireless/ethernet card just broke. Maybe you don’t have access to a shared server for the moment, you want to email someone updates and you don’t want to transfer 40 commits via `format-patch`.

This is where the `git bundle` command can be helpful. The `bundle` command will package up everything that would normally be pushed over the wire with a `git push` command into a binary file that you can email to someone or put on a flash drive, then unbundle into another repository.

Let’s see a simple example. Let’s say you have a repository with two commits:

```
$ git log
commit 9a466c572fe88b195efd356c3f2bbeccdb504102
Author: Scott Chacon <schacon@gmail.com>
Date:   Wed Mar 10 07:34:10 2010 -0800

    second commit

commit b1ec3248f39900d2a406049d762aa68e9641be25
Author: Scott Chacon <schacon@gmail.com>
Date:   Wed Mar 10 07:34:01 2010 -0800

    first commit
```

If you want to send that repository to someone and you don’t have access to a repository to push to, or simply don’t want to set one up, you can bundle it with `git bundle create`.

```
$ git bundle create repo.bundle HEAD master
Counting objects: 6, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (6/6), 441 bytes, done.
Total 6 (delta 0), reused 0 (delta 0)
```

Now you have a file named `repo.bundle` that has all the data needed to recreate the repository’s `master` branch. With the `bundle` command you need to list out every reference or specific range of commits that you want to be includ-

ed. If you intend for this to be cloned somewhere else, you should add HEAD as a reference as well as we've done here.

You can email this `repo.bundle` file to someone else, or put it on a USB drive and walk it over.

On the other side, say you are sent this `repo.bundle` file and want to work on the project. You can clone from the binary file into a directory, much like you would from a URL.

```
$ git clone repo.bundle repo
Initialized empty Git repository in /private/tmp/bundle/repo/.git/
$ cd repo
$ git log --oneline
9a466c5 second commit
b1ec324 first commit
```

If you don't include HEAD in the references, you have to also specify `-b master` or whatever branch is included because otherwise it won't know what branch to check out.

Now let's say you do three commits on it and want to send the new commits back via a bundle on a USB stick or email.

```
$ git log --oneline
71b84da last commit - second repo
c99cf5b fourth commit - second repo
7011d3d third commit - second repo
9a466c5 second commit
b1ec324 first commit
```

First we need to determine the range of commits we want to include in the bundle. Unlike the network protocols which figure out the minimum set of data to transfer over the network for us, we'll have to figure this out manually. Now, you could just do the same thing and bundle the entire repository, which will work, but it's better to just bundle up the difference - just the three commits we just made locally.

In order to do that, you'll have to calculate the difference. As we described in “[Commit Ranges](#)”, you can specify a range of commits in a number of ways. To get the three commits that we have in our master branch that weren't in the branch we originally cloned, we can use something like `origin/master..master` or `master ^origin/master`. You can test that with the `log` command.

```
$ git log --oneline master ^origin/master
71b84da last commit - second repo
c99cf5b fourth commit - second repo
7011d3d third commit - second repo
```

So now that we have the list of commits we want to include in the bundle, let's bundle them up. We do that with the `git bundle create` command, giving it a filename we want our bundle to be and the range of commits we want to go into it.

```
$ git bundle create commits.bundle master ^9a466c5
Counting objects: 11, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (9/9), 775 bytes, done.
Total 9 (delta 0), reused 0 (delta 0)
```

Now we have a `commits.bundle` file in our directory. If we take that and send it to our partner, she can then import it into the original repository, even if more work has been done there in the meantime.

When she gets the bundle, she can inspect it to see what it contains before she imports it into her repository. The first command is the `bundle verify` command that will make sure the file is actually a valid Git bundle and that you have all the necessary ancestors to reconstitute it properly.

```
$ git bundle verify ../commits.bundle
The bundle contains 1 ref
71b84daaf49abed142a373b6e5c59a22dc6560dc refs/heads/master
The bundle requires these 1 ref
9a466c572fe88b195efdf356c3f2bbeccdb504102 second commit
../commits.bundle is okay
```

If the bundler had created a bundle of just the last two commits they had done, rather than all three, the original repository would not be able to import it, since it is missing requisite history. The `verify` command would have looked like this instead:

```
$ git bundle verify ../commits-bad.bundle
error: Repository lacks these prerequisite commits:
error: 7011d3d8fc200abe0ad561c011c3852a4b7bbe95 third commit - second repo
```

However, our first bundle is valid, so we can fetch in commits from it. If you want to see what branches are in the bundle that can be imported, there is also a command to just list the heads:

```
$ git bundle list-heads ../commits.bundle
71b84daaf49abed142a373b6e5c59a22dc6560dc refs/heads/master
```

The `verify` sub-command will tell you the heads as well. The point is to see what can be pulled in, so you can use the `fetch` or `pull` commands to import commits from this bundle. Here we'll fetch the `master` branch of the bundle to a branch named `other-master` in our repository:

```
$ git fetch ../commits.bundle master:other-master
From ../commits.bundle
 * [new branch] master -> other-master
```

Now we can see that we have the imported commits on the `other-master` branch as well as any commits we've done in the meantime in our own `master` branch.

```
$ git log --oneline --decorate --graph --all
* 8255d41 (HEAD, master) third commit - first repo
| * 71b84da (other-master) last commit - second repo
| * c99cf5b fourth commit - second repo
| * 7011d3d third commit - second repo
|
* 9a466c5 second commit
* b1ec324 first commit
```

So, `git bundle` can be really useful for sharing or doing network-type operations when you don't have the proper network or shared repository to do so.

Replace

Git's objects are unchangeable, but it does provide an interesting way to pretend to replace objects in its database with other objects.

The `replace` command lets you specify an object in Git and say “every time you see this, pretend it's this other thing”. This is most commonly useful for replacing one commit in your history with another one.

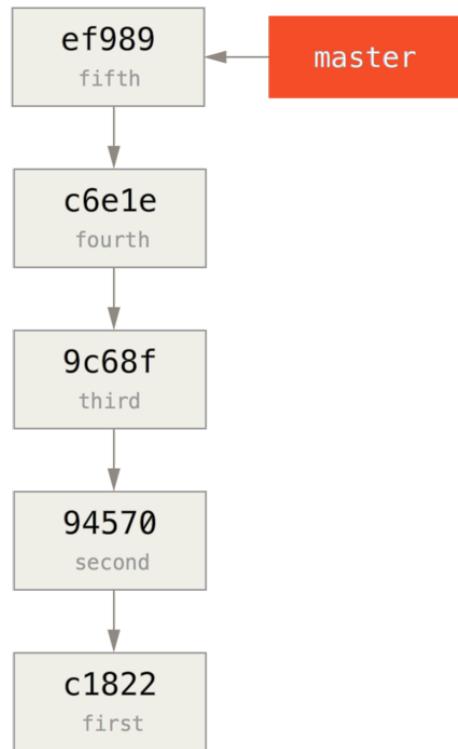
For example, let's say you have a huge code history and want to split your repository into one short history for new developers and one much longer and larger history for people interested in data mining. You can graft one history onto the other by `replace`ing the earliest commit in the new line with the latest commit on the older one. This is nice because it means that you don't actually have to rewrite every commit in the new history, as you would normally have to do to join them together (because the parentage effects the SHA-1s).

Let's try this out. Let's take an existing repository, split it into two repositories, one recent and one historical, and then we'll see how we can recombine them without modifying the recent repositories SHA-1 values via `replace`.

We'll use a simple repository with five simple commits:

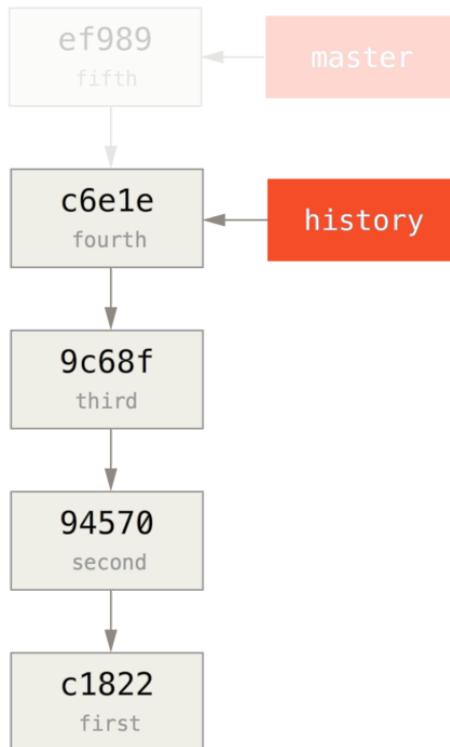
```
$ git log --oneline
ef989d8 fifth commit
c6e1e95 fourth commit
9c68fdc third commit
945704c second commit
c1822cf first commit
```

We want to break this up into two lines of history. One line goes from commit one to commit four - that will be the historical one. The second line will just be commits four and five - that will be the recent history.

FIGURE 7-28

Well, creating the historical history is easy, we can just put a branch in the history and then push that branch to the master branch of a new remote repository.

```
$ git branch history c6e1e95
$ git log --oneline --decorate
ef989d8 (HEAD, master) fifth commit
c6e1e95 (history) fourth commit
9c68fdc third commit
945704c second commit
c1822cf first commit
```

FIGURE 7-29

Now we can push the new `history` branch to the `master` branch of our new repository:

```
$ git remote add project-history https://github.com/schacon/project-history
$ git push project-history history:master
Counting objects: 12, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (12/12), 907 bytes, done.
Total 12 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (12/12), done.
To git@github.com:schacon/project-history.git
 * [new branch]      history -> master
```

OK, so our history is published. Now the harder part is truncating our recent history down so it's smaller. We need an overlap so we can replace a commit in one with an equivalent commit in the other, so we're going to truncate this to just commits four and five (so commit four overlaps).

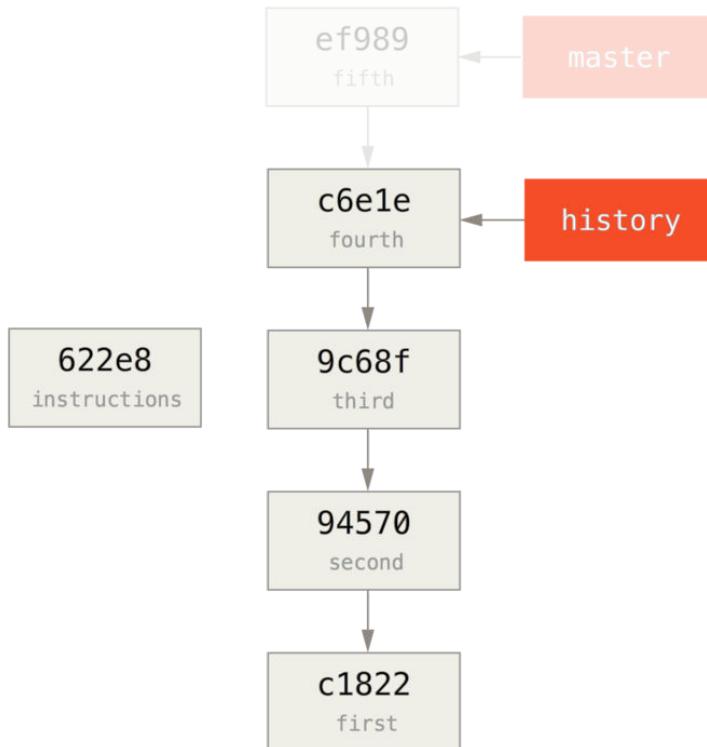
```
$ git log --oneline --decorate
ef989d8 (HEAD, master) fifth commit
c6e1e95 (history) fourth commit
9c68fdc third commit
945704c second commit
c1822cf first commit
```

It's useful in this case to create a base commit that has instructions on how to expand the history, so other developers know what to do if they hit the first commit in the truncated history and need more. So, what we're going to do is create an initial commit object as our base point with instructions, then rebase the remaining commits (four and five) on top of it.

To do that, we need to choose a point to split at, which for us is the third commit, which is 9c68fdc in SHA-speak. So, our base commit will be based off of that tree. We can create our base commit using the `commit-tree` command, which just takes a tree and will give us a brand new, parentless commit object SHA-1 back.

```
$ echo 'get history from blah blah blah' | git commit-tree 9c68fdc^{tree}
622e88e9cbfbacfb75b5279245b9fb38dfa10cf
```

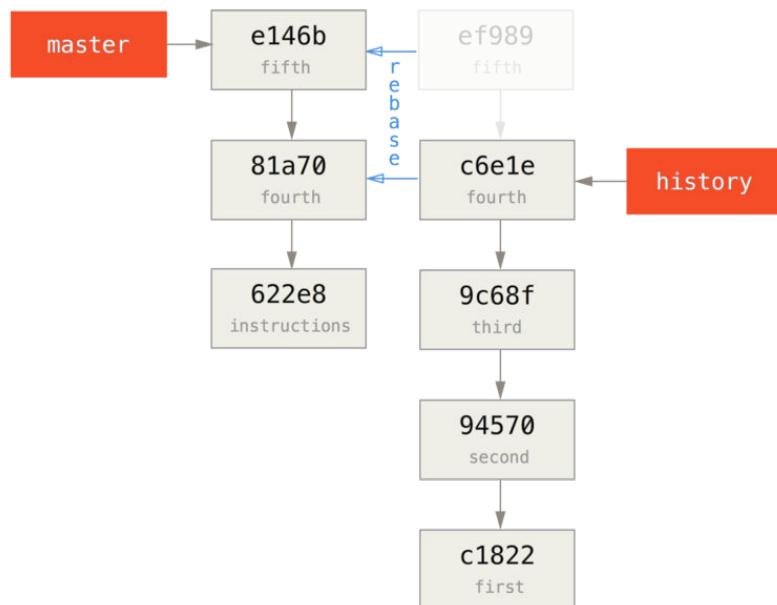
The `commit-tree` command is one of a set of commands that are commonly referred to as *plumbing* commands. These are commands that are not generally meant to be used directly, but instead are used by `git` commands to do smaller jobs. On occasions when we're doing weirder things like this, they allow us to do really low-level things but are not meant for daily use. You can read more about plumbing commands in “[Plumbing and Porcelain](#)”

FIGURE 7-30

OK, so now that we have a base commit, we can rebase the rest of our history on top of that with `git rebase --onto`. The `--onto` argument will be the SHA-1 we just got back from `commit-tree` and the rebase point will be the third commit (the parent of the first commit we want to keep, 9c68fdc):

```
$ git rebase --onto 622e88 9c68fdc
First, rewinding head to replay your work on top of it...
Applying: fourth commit
Applying: fifth commit
```

FIGURE 7-31



OK, so now we've re-written our recent history on top of a throw away base commit that now has instructions in it on how to reconstitute the entire history if we wanted to. We can push that new history to a new project and now when people clone that repository, they will only see the most recent two commits and then a base commit with instructions.

Let's now switch roles to someone cloning the project for the first time who wants the entire history. To get the history data after cloning this truncated repository, one would have to add a second remote for the historical repository and fetch:

```
$ git clone https://github.com/schacon/project
$ cd project

$ git log --oneline master
e146b5f fifth commit
81a708d fourth commit
622e88e get history from blah blah blah

$ git remote add project-history https://github.com/schacon/project-history
$ git fetch project-history
```

```
From https://github.com/schacon/project-history
 * [new branch]      master      -> project-history/master
```

Now the collaborator would have their recent commits in the `master` branch and the historical commits in the `project-history/master` branch.

```
$ git log --oneline master
e146b5f fifth commit
81a708d fourth commit
622e88e get history from blah blah blah

$ git log --oneline project-history/master
c6e1e95 fourth commit
9c68fdc third commit
945704c second commit
c1822cf first commit
```

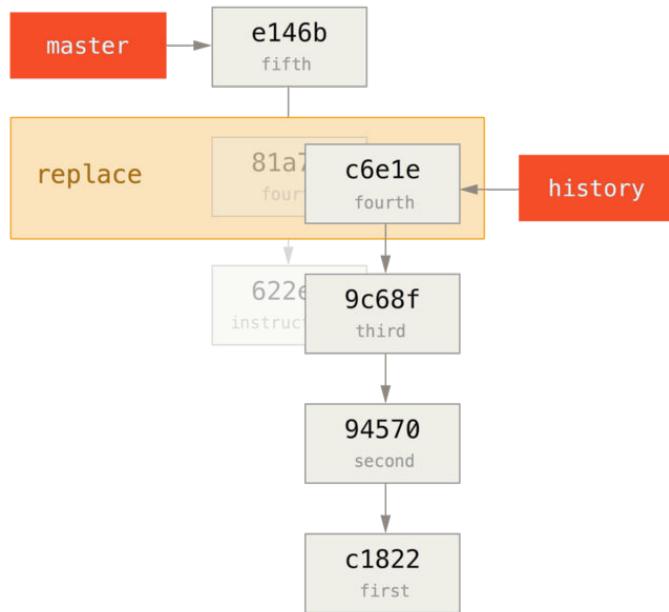
To combine them, you can simply call `git replace` with the commit you want to replace and then the commit you want to replace it with. So we want to replace the “fourth” commit in the `master` branch with the “fourth” commit in the `project-history/master` branch:

```
$ git replace 81a708d c6e1e95
```

Now, if you look at the history of the `master` branch, it appears to look like this:

```
$ git log --oneline master
e146b5f fifth commit
81a708d fourth commit
9c68fdc third commit
945704c second commit
c1822cf first commit
```

Cool, right? Without having to change all the SHA-1s upstream, we were able to replace one commit in our history with an entirely different commit and all the normal tools (`bisect`, `blame`, etc) will work how we would expect them to.

FIGURE 7-32

Interestingly, it still shows `81a708d` as the SHA-1, even though it's actually using the `c6e1e95` commit data that we replaced it with. Even if you run a command like `cat-file`, it will show you the replaced data:

```
$ git cat-file -p 81a708d
tree 7bc544cf438903b65ca9104a1e30345eee6c083d
parent 9c68fdceee073230f19ebb8b5e7fc71b479c0252
author Scott Chacon <schacon@gmail.com> 1268712581 -0700
committer Scott Chacon <schacon@gmail.com> 1268712581 -0700

fourth commit
```

Remember that the actual parent of `81a708d` was our placeholder commit (`622e88e`), not `9c68fdce` as it states here.

Another interesting thing is that this data is kept in our references:

```
$ git for-each-ref
e146b5f14e79d4935160c0e83fb9ebe526b8da0d commit refs/heads/master
c6e1e95051d41771a649f3145423f8809d1a74d4 commit refs/remotes/history/master
```

```
e146b5f14e79d4935160c0e83fb9ebe526b8da0d commit refs/remotes/origin/HEAD  
e146b5f14e79d4935160c0e83fb9ebe526b8da0d commit refs/remotes/origin/master  
c6e1e95051d41771a649f3145423f8809d1a74d4 commit refs/replace/81a708dd0e167a3f691541c7a646334
```

This means that it's easy to share our replacement with others, because we can push this to our server and other people can easily download it. This is not that helpful in the history grafting scenario we've gone over here (since everyone would be downloading both histories anyhow, so why separate them?) but it can be useful in other circumstances.

Credential Storage

If you use the SSH transport for connecting to remotes, it's possible for you to have a key without a passphrase, which allows you to securely transfer data without typing in your username and password. However, this isn't possible with the HTTP protocols – every connection needs a username and password. This gets even harder for systems with two-factor authentication, where the token you use for a password is randomly generated and unpronounceable.

Fortunately, Git has a credentials system that can help with this. Git has a few options provided in the box:

- The default is not to cache at all. Every connection will prompt you for your username and password.
- The “cache” mode keeps credentials in memory for a certain period of time. None of the passwords are ever stored on disk, and they are purged from the cache after 15 minutes.
- The “store” mode saves the credentials to a plain-text file on disk, and they never expire. This means that until you change your password for the Git host, you won’t ever have to type in your credentials again. The downside of this approach is that your passwords are stored in cleartext in a plain file in your home directory.
- If you’re using a Mac, Git comes with an “osxkeychain” mode, which caches credentials in the secure keychain that’s attached to your system account. This method stores the credentials on disk, and they never expire, but they’re encrypted with the same system that stores HTTPS certificates and Safari auto-fills.
- If you’re using Windows, you can install a helper called “winstore.” This is similar to the “osxkeychain” helper described above, but uses the Windows Credential Store to control sensitive information. It can be found at <https://gitcredentialstore.codeplex.com>.

You can choose one of these methods by setting a Git configuration value:

```
$ git config --global credential.helper cache
```

Some of these helpers have options. The “store” helper can take a `--file <path>` argument, which customizes where the plaintext file is saved (the default is `~/ .git-credentials`). The “cache” helper accepts the `--timeout <seconds>` option, which changes the amount of time its daemon is kept running (the default is “900”, or 15 minutes). Here’s an example of how you’d configure the “store” helper with a custom file name:

```
$ git config --global credential.helper store --file ~/my-credentials
```

Git even allows you to configure several helpers. When looking for credentials for a particular host, Git will query them in order, and stop after the first answer is provided. When saving credentials, Git will send the username and password to **all** of the helpers in the list, and they can choose what to do with them. Here’s what a `.gitconfig` would look like if you had a credentials file on a thumb drive, but wanted to use the in-memory cache to save some typing if the drive isn’t plugged in:

```
[credential]
helper = store --file /mnt/thumbdrive/.git-credentials
helper = cache --timeout 30000
```

Under the Hood

How does this all work? Git’s root command for the credential-helper system is `git credential`, which takes a command as an argument, and then more input through `stdin`.

This might be easier to understand with an example. Let’s say that a credential helper has been configured, and the helper has stored credentials for `mygithost`. Here’s a session that uses the “fill” command, which is invoked when Git is trying to find credentials for a host:

```
$ git credential fill ❶
protocol=https ❷
host=mygithost
❸
protocol=https ❹
```

```

host=mygithost
username=bob
password=s3cre7
$ git credential fill ⑤
protocol=https
host=unknownhost

Username for 'https://unknownhost': bob
Password for 'https://bob@unknownhost':
protocol=https
host=unknownhost
username=bob
password=s3cre7

```

- ① This is the command line that initiates the interaction.
- ② Git-credential is then waiting for input on stdin. We provide it with the things we know: the protocol and hostname.
- ③ A blank line indicates that the input is complete, and the credential system should answer with what it knows.
- ④ Git-credential then takes over, and writes to stdout with the bits of information it found.
- ⑤ If credentials are not found, Git asks the user for the username and password, and provides them back to the invoking stdout (here they're attached to the same console).

The credential system is actually invoking a program that's separate from Git itself; which one and how depends on the `credential.helper` configuration value. There are several forms it can take:

| Configuration Value | Behavior |
|--|---|
| foo | Runs <code>git-credential-foo</code> |
| foo -a --opt=bcd | Runs <code>git-credential-foo -a --opt=bcd</code> |
| /absolute/path/foo -xyz | Runs <code>/absolute/path/foo -xyz</code> |
| <code>!f() { echo "password=s3cre7"; }; f</code> | Code after ! evaluated in shell |

So the helpers described above are actually named `git-credential-cache`, `git-credential-store`, and so on, and we can configure them to take command-line arguments. The general form for this is “`git-credential-foo [args] <action>`.” The `stdin/stdout` protocol is the same as `git-credential`, but they use a slightly different set of actions:

- `get` is a request for a username/password pair.
- `store` is a request to save a set of credentials in this helper’s memory.
- `erase` purge the credentials for the given properties from this helper’s memory.

For the `store` and `erase` actions, no response is required (Git ignores it anyway). For the `get` action, however, Git is very interested in what the helper has to say. If the helper doesn’t know anything useful, it can simply exit with no output, but if it does know, it should augment the provided information with the information it has stored. The output is treated like a series of assignment statements; anything provided will replace what Git already knows.

Here’s the same example from above, but skipping `git-credential` and going straight for `git-credential-store`:

```
$ git credential-store --file ~/git.store store ❶
protocol=https
host=mygithost
username=bob
password=s3cre7
$ git credential-store --file ~/git.store get ❷
protocol=https
host=mygithost

username=bob ❸
password=s3cre7
```

- ❶ Here we tell `git-credential-store` to save some credentials: the username “bob” and the password “s3cre7” are to be used when `https://mygithost` is accessed.
- ❷ Now we’ll retrieve those credentials. We provide the parts of the connection we already know (`https://mygithost`), and an empty line.
- ❸ `git-credential-store` replies with the username and password we stored above.

Here’s what the `~/git.store` file looks like:

```
https://bob:s3cre7@mygithost
```

It's just a series of lines, each of which contains a credential-decorated URL. The `osxkeychain` and `winstore` helpers use the native format of their backing stores, while `cache` uses its own in-memory format (which no other process can read).

A Custom Credential Cache

Given that `git-credential-store` and friends are separate programs from Git, it's not much of a leap to realize that *any* program can be a Git credential helper. The helpers provided by Git cover many common use cases, but not all. For example, let's say your team has some credentials that are shared with the entire team, perhaps for deployment. These are stored in a shared directory, but you don't want to copy them to your own credential store, because they change often. None of the existing helpers cover this case; let's see what it would take to write our own. There are several key features this program needs to have:

1. The only action we need to pay attention to is `get`; `store` and `erase` are write operations, so we'll just exit cleanly when they're received.
2. The file format of the shared-credential file is the same as that used by `git-credential-store`.
3. The location of that file is fairly standard, but we should allow the user to pass a custom path just in case.

Once again, we'll write this extension in Ruby, but any language will work so long as Git can execute the finished product. Here's the full source code of our new credential helper:

```
#!/usr/bin/env ruby

require 'optparse'

path = File.expand_path '~/.git-credentials' ❶
OptionParser.new do |opts|
  opts.banner = 'USAGE: git-credential-read-only [options] <action>'
  opts.on('-f', '--file PATH', 'Specify path for backing store') do |argpath|
    path = File.expand_path argpath
  end
end.parse!

exit(0) unless ARGV[0].downcase == 'get' ❷
exit(0) unless File.exists? path
```

```

known = {} ❸
while line = STDIN.gets
  break if line.strip == ''
  k,v = line.strip.split '=', 2
  known[k] = v
end

File.readlines(path).each do |fileline| ❹
  prot,user,pass,host = fileline.scan(/^(.*?):\/\/(.*):(.*?)@(.*)$/).first
  if prot == known['protocol'] and host == known['host']
    puts "protocol=#{prot}"
    puts "host=#{host}"
    puts "username=#{user}"
    puts "password=#{pass}"
    exit(0)
  end
end

```

- ❶ Here we parse the command-line options, allowing the user to specify the input file. The default is `~/.git-credentials`.
- ❷ This program only responds if the action is `get` and the backing-store file exists.
- ❸ This loop reads from `stdin` until the first blank line is reached. The inputs are stored in the `known` hash for later reference.
- ❹ This loop reads the contents of the storage file, looking for matches. If the protocol and host from `known` match this line, the program prints the results to `stdout` and exits.

We'll save our helper as `git-credential-read-only`, put it somewhere in our `PATH` and mark it executable. Here's what an interactive session looks like:

```
$ git credential-read-only --file=/mnt/shared/creds get
protocol=https
host=mygithost

protocol=https
host=mygithost
username=bob
password=s3cre7
```

Since its name starts with “git-”, we can use the simple syntax for the configuration value:

```
$ git config --global credential.helper read-only --file /mnt/shared/creds
```

As you can see, extending this system is pretty straightforward, and can solve some common problems for you and your team.

Summary

You’ve seen a number of advanced tools that allow you to manipulate your commits and staging area more precisely. When you notice issues, you should be able to easily figure out what commit introduced them, when, and by whom. If you want to use subprojects in your project, you’ve learned how to accommodate those needs. At this point, you should be able to do most of the things in Git that you’ll need on the command line day to day and feel comfortable doing so.

Personalización de Git

Hasta ahora, hemos visto los aspectos básicos del funcionamiento de Git y la manera de utilizarlo; además de haber presentado una serie de herramientas suministradas con Git para ayudarnos a usarlo de manera sencilla y eficiente. En este capítulo, avanzaremos sobre ciertas operaciones que puedes utilizar para personalizar el funcionamiento de Git ; presentando algunos de sus principales ajustes y el sistema de anclajes (hooks). Con estas operaciones, será fácil conseguir que Git trabaje exactamente como tú, tu empresa o tu grupo necesitáis.

Configuración de Git

Como se ha visto brevemente en **Chapter 1**, podemos acceder a los ajustes de configuración de Git a través del comando *git config*. Una de las primeras acciones que has realizado con Git ha sido el configurar tu nombre y tu dirección de correo-e.

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

Ahora vas a aprender un puñado de nuevas e interesantes opciones que puedes utilizar para personalizar el uso de Git.

Primeramente, vamos a repasar brevemente los detalles de configuración de Git que ya has visto. Para determinar su comportamiento no estandar, Git emplea una serie de archivos de configuración. El primero de ellos es el archivo */etc/gitconfig*, que contiene valores para todos y cada uno de los usuarios en el sistema y para todos sus repositorios. Con la opción *--system* del comando *git config*, puedes leer y escribir de/a este archivo.

El segundo es el archivo `~/.gitconfig` (o `~/.config/git/config`), específico para cada usuario. Con la opción `--global`, `git config` lee y escribe en este archivo.

Y por último, Git también puede considerar valores de configuración presentes en el archivo `.git/config` de cada repositorio que estés utilizando. Estos valores se aplicarán únicamente a dicho repositorio.

Cada nivel sobreescribe los valores del nivel anterior; es decir lo configurado en `.git/config` tiene preferencia con respecto a lo configurado en `/etc/gitconfig`, por ejemplo.

Los ficheros de configuración de Git son de texto plano, por lo que también puedes ajustar manualmente los valores de configuración, editando directamente los archivos correspondientes y escribiendo en ellos con la sintaxis correspondiente; pero suele ser más sencillo hacerlo siempre a través del comando `git config`.

Configuración básica del cliente

Las opciones de configuración reconocidas por Git pueden distribuirse en dos grandes categorías: las del lado cliente y las del lado servidor. La mayoría de las opciones están en el lado cliente, – configurando tus preferencias personales de trabajo -. Aunque hay multitud de ellas, aquí vamos a ver solamente unas pocas, las más comúnmente utilizadas o las que afectan significativamente a tu forma de trabajar. No vamos a revisar aquellas opciones utilizadas solo en casos muy especiales. Si quieres consultar una lista completa, con todas las opciones contempladas en tu versión de Git, puedes lanzar el comando:

```
$ man git-config
```

Este comando muestra todas las opciones con cierto nivel de detalle. También puedes encontrar esta información de referencia en <http://git-scm.com/docs/git-config.html>.

CORE.EDITOR

Por defecto, Git utiliza cualquier editor que hayas configurado como editor de texto por defecto de tu sistema (`$VISUAL` o `$EDITOR`). O, si no lo has configurado, utilizará `vi` como editor para crear y editar las etiquetas y mensajes de tus confirmaciones de cambio (commit). Para cambiar ese comportamiento, puedes utilizar el ajuste `core.editor`:

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

A partir de ese comando, por ejemplo, git lanzará Emacs cada vez que vaya a editar mensajes; indistintamente del editor configurado en la línea de comandos (shell) del sistema.

COMMIT TEMPLATE

Si preparas este ajuste para apuntar a un archivo concreto de tu sistema, Git lo utilizará como mensaje por defecto cuando hagas confirmaciones de cambio. Por ejemplo, imagina que creas una plantilla en `~/.gitmessage.txt` con un contenido tal como:

```
subject line
```

```
what happened
```

```
[ticket: X]
```

Para indicar a Git que lo utilice como mensaje por defecto y que aparezca en tu editor cuando lances el comando `git commit`, tan solo has de ajustar `commit.template`:

```
$ git config --global commit.template ~/.gitmessage.txt
$ git commit
```

A partir de entonces, cada vez que confirmes cambios (commit), tu editor se abrirá con algo como esto:

```
subject line
```

```
what happened
```

```
[ticket: X]
```

```
# Please enter the commit message for your changes. Lines starting
```

```
# with '#' will be ignored, and an empty message aborts the commit.
```

```
# On branch master
```

```
# Changes to be committed:
```

```
#   (use "git reset HEAD <file>..." to unstage)
```

```
#
```

```
# modified: lib/test.rb
```

```
#
```

```
~
```

```
~  
.git/COMMIT_EDITMSG" 14L, 297C
```

Si tienes una política concreta con respecto a los mensajes de confirmación de cambios, puedes aumentar las posibilidades de que sea respetada si creas una plantilla acorde a dicha política y la pones como plantilla por defecto de Git.

CORE.PAGER

El parámetro `core.pager` selecciona el paginador utilizado por Git cuando muestra resultados de comandos tales como `log` o `diff..`. Puedes ajustarlo para que utilice `more` o tu paginador favorito, (por defecto, se utiliza `less`); o puedes anular la paginación si le asignas una cadena vacía.

```
$ git config --global core.pager ''
```

Si lanzas esto, Git mostrará siempre el resultado completo de todos los mandos, independientemente de lo largo que sea éste.

USER.SIGNINGKEY

Si tienes costumbre de firmar tus etiquetas (tal y como se ha visto en “[Signing Your Work](#)”), configurar tu clave de firma GPG puede facilitarte la labor. Puedes configurar tu clave ID de esta forma:

```
$ git config --global user.signingkey <gpg-key-id>
```

Ahora podrás firmar etiquetas sin necesidad de indicar tu clave cada vez en el comando `git tag`.

```
$ git tag -s <tag-name>
```

CORE.EXCLUDESFILE

Se pueden indicar expresiones en el archivo `.gitignore` de tu proyecto para indicar a Git lo que debe considerar o no como archivos sin seguimiento, o lo que interará o no seleccionar cuando lances el comando `git add`, tal y como se indicó en “[Ignorar Archivos](#)”.

Pero a veces, necesitas ignorar ciertos ficheros en todos los repositorios con los que trabajas. Por ejemplo, si trabajas en una computadora con Mac OS X, estarás al tanto de la existencia de los fichero `.DS_Store`. O si usas Emacs o Vim, también conocerás los ficheros terminados en `~`.

Este ajuste puedes grabarlo en un fichero global, llamado `~/gitignore_global`, con estos contenidos:

```
*~
.DS_Store
```

...y si ahora lanzas `git config --global core.excludesfile ~/gitignore_global`, Git ya nunca más tendrá en cuenta esos ficheros en tus repositorios.

HELP.AUTOCORRECT

Si te equivocas al teclear un comando de Git, te mostrará algo como:

```
$ git chekcout master
git: 'chekcout' is not a git command. See 'git --help'.

Did you mean this?
  checkout
```

Git intenta imaginar qué es lo que querías escribir, pero aun así no lo intenta ejecutar. Si pones la opción `help.autocorrect` a 1, Git sí lanzará el comando corrigiendo tu error:

```
$ git chekcout master
WARNING: You called a Git command named 'chekcout', which does not exist.
Continuing under the assumption that you meant 'checkout'
in 0.1 seconds automatically...
```

Observa lo de “0.1 seconds”. Es un entero que representa décimas de segundo. Si le das un valor 50, Git retrasará la ejecución final del comando 5 segundos con el fin de que puedas abortar la operación auto-corregida con la opción `help.autocorrect`.

Colores en Git

Git puede marcar con colores los resultados que muestra en tu terminal, ayudandote así a leerlos más fácilmente. Hay unos cuantos parámetros que te pueden ayudar a configurar tus colores favoritos.

COLOR.UI

Si se lo pides, Git coloreará automáticamente la mayor parte de los resultados que muestre. Puedes ajustar con precisión cada una de las partes a colorear; pero si deseas activar de un golpe todos los colores por defecto, no tienes más que poner a “true” el parámetro `color.ui`.

Para desactivar totalmente los colores, puedes hacer esto:

```
$ git config --global color.ui false
```

El valor predeterminado es `auto`, que colorea la salida cuando va a un terminal, pero no lo hace cuando se envía la salida a un fichero o a una tubería.

También puedes ponerlo a `always` para hacer que se coloree siempre. Es muy raro que quieras hacer esto, ya que cuando se quiere puntualmente colorear la salida redirigida se puede pasar un flag `--color` al comando Git.

COLOR.*

Cuando quieras ajustar específicamente, comando a comando, donde colorear y cómo colorear, puedes emplear los ajustes particulares de color. Cada uno de ellos puede fijarse a `true` (verdadero), `false` (falso) o `always` (siempre):

```
color.branch
color.diff
color.interactive
color.status
```

Además, cada uno de ellos tiene parámetros adicionales para asignar colores a partes específicas, por si quieras precisar aún más. Por ejemplo, para mostrar la meta-information del comando `diff` con letra azul sobre fondo negro y con caracteres en negrita, puedes indicar:

```
$ git config --global color.diff.meta "blue black bold"
```

Puedes ajustar un color a cualquiera de los siguientes valores: `normal`, `black` (negro), `red` (rojo), `green` (verde), `yellow` (amarillo), `blue` (azul), `magenta`, `cyan` (cian), o `white` (blanco).

También puedes aplicar atributos tales como `bold` (negrilla), `dim` (tenue), `ul` (subrayado), `blink` (parpadeante) y `reverse` (video inverso).

Herramientas externas para fusión y diferencias

Aunque Git lleva una implementación interna de diff, la que se utiliza habitualmente, se puede sustituir por una herramienta externa. Puedes incluso configurar una herramienta gráfica para la resolución de conflictos, en lugar de resolverlos manualmente. Lo voy a demostrar configurando Perforce Visual Merge (P4Merge) como herramienta para realizar las comparaciones y resolver conflictos; ya que es una buena herramienta gráfica y es libre.

Si lo quieres probar, P4Merge funciona en todas las principales plataformas. Los nombres de carpetas que utilizaré en los ejemplos funcionan en sistemas Mac y Linux; para Windows, tendrás que sustituir `/usr/local/bin` por el correspondiente camino al ejecutable en tu sistema.

P4Merge se puede descargar desde <http://www.perforce.com/downloads/Perforce/>.

Para empezar, tienes que preparar los correspondientes scripts para lanzar tus comandos. En estos ejemplos, voy a utilizar caminos y nombres Mac para los ejecutables; en otros sistemas, tendrás que sustituirlos por los correspondientes donde tengas instalado `p4merge`. El primer script a preparar es uno al que denominaremos `extMerge`, para llamar al ejecutable con los correspondientes argumentos:

```
$ cat /usr/local/bin/extMerge
#!/bin/sh
/Applications/p4merge.app/Contents/MacOS/p4merge $*
```

El script para el comparador, ha de asegurarse de recibir siete argumentos y de pasar dos de ellos al script de fusion (merge). Por defecto, Git pasa los siguientes argumentos al programa diff (comparador):

```
path old-file old-hex old-mode new-file new-hex new-mode
```

Ya que solo necesitarás `old-file` y `new-file`, puedes utilizar el siguiente script para extraerlos:

```
$ cat /usr/local/bin/extDiff
#!/bin/sh
[ $# -eq 7 ] && /usr/local/bin/extMerge "$2" "$5"
```

Además has de asegurarte de que estas herramientas son ejecutables:

```
$ sudo chmod +x /usr/local/bin/extMerge
$ sudo chmod +x /usr/local/bin/extDiff
```

Una vez preparado todo esto, puedes ajustar el archivo de configuración para utilizar tus herramientas personalizadas de comparación y resolución de conflictos. Tenemos varios parámetros a ajustar: `merge.tool` para indicar a Git la estrategia que ha de usar, `mergetool.*.cmd` para especificar como lanzar el comando, `mergetool.trustExitCode` para decir a Git si el código de salida del programa indica una fusión con éxito o no, y `diff.external` para decir a Git qué comando lanzar para realizar comparaciones. Es decir, has de ejecutar cuatro comandos de configuración:

```
$ git config --global merge.tool extMerge
$ git config --global mergetool.extMerge.cmd \
    'extMerge \"$BASE\" \"$LOCAL\" \"$REMOTE\" \"$MERGED\"'
$ git config --global mergetool.extMerge.trustExitCode false
$ git config --global diff.external extDiff
```

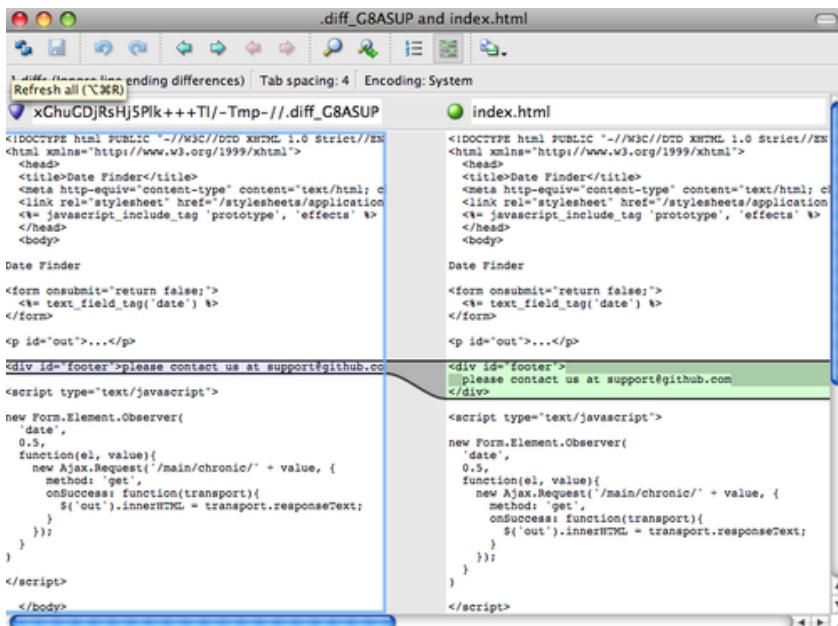
o puedes editar tu archivo `~/.gitconfig` para añadirle las siguientes líneas:

```
[merge]
  tool = extMerge
[mergetool "extMerge"]
  cmd = extMerge "$BASE" "$LOCAL" "$REMOTE" "$MERGED"
  trustExitCode = false
[diff]
  external = extDiff
```

Tras ajustar todo esto, si lanzas comandos tales como:

```
$ git diff 32d1776b1^ 32d1776b1
```

En lugar de mostrar las diferencias por línea de comandos, Git lanzará P4Merge, que tiene una pinta como:

**FIGURE 8-1**

P4Merge.

Si intentas fusionar (merge) dos ramas y tienes los consabidos conflictos de integración, puedes lanzar el comando `git mergetool`; lanzará P4Merge para ayudarte a resolver los conflictos por medio de su interfaz gráfica.

Lo bonito de estos ajustes con scripts, es que puedes cambiar fácilmente tus herramientas de comparación (diff) y de fusión (merge). Por ejemplo, para cambiar tus scripts `extDiff` y `extMerge` para utilizar KDiff3, tan solo has de editar el archivo `extMerge`:

```
$ cat /usr/local/bin/extMerge
#!/bin/sh
/Applications/kdiff3.app/Contents/MacOS/kdiff3 $*
```

A partir de ahora, Git utilizará la herramienta KDiff3 para mostrar y resolver conflictos de integración.

Git viene preparado para utilizar bastantes otras herramientas de resolución de conflictos, sin necesidad de andar ajustando la configuración. Para listar las herramientas soportadas solo has de lanzar el comando:

```
$ git mergetool --tool-help
'git mergetool --tool=<tool>' may be set to one of the following:
```

```
emerge
gvimdiff
gvimdiff2
opendiff
p4merge
vimdiff
vimdiff2
```

The following tools are valid, but not currently available:

```
araxis
bc3
codecompare
deltawalker
diffmerge
diffuse
ecmerge
kdiff3
meld
tkdiff
tortoisemerge
xxdiff
```

Some of the tools listed above only work in a windowed environment. If run in a terminal-only session, they will fail.

Si no te interesa utilizar KDiff3 para comparaciones, sino que tan solo te interesa utilizarlo para resolver conflictos de integración; teniendo kdiff3 en el path de ejecución, solo has de lanzar el comando:

```
$ git config --global merge.tool kdiff3
```

Si utilizas este comando en lugar de preparar los archivos extMerge y extDiff antes comentados, Git utilizará KDiff3 para resolución de conflictos de integración y la herramienta estandar diff para las comparaciones.

Formateo y espacios en blanco

El formato y los espacios en blanco son la fuente de los problemas más sutiles y frustrantes que muchos desarrolladores se pueden encontrar en entornos colaborativos, especialmente si son multi-plataforma. Es muy facil que algunos parches u otro trabajo recibido introduzcan sutiles cambios de espaciado, porque los editores suelen hacerlo inadvertidamente o, trabajando en entornos multi-plataforma, porque programadores Windows suelen añadir retornos de

carro al final de las líneas que tocan. Git dispone de algunas opciones de configuración para ayudarnos con estos problemas.

CORE.AUTOCRLF

Si estás programando en Windows o utilizando algún otro sistema, pero colaborando con gente que programa en Windows. Es muy posible que alguna vez te topes con problemas de finales de línea. Esto se debe a que Windows utiliza retorno-de-carro y salto-de-línea para marcar los finales de línea de sus archivos. Mientras que Mac y Linux utilizan solamente el carácter de salto-de-línea. Esta es una útil, pero molesta, diferencia cuando se trabaja en entornos multiplataforma.

Git la maneja convirtiendo automáticamente los finales CRLF en LF al hacer confirmaciones de cambios (commit); y, viceversa, al extraer código (checkout) a la carpeta de trabajo. Puedes activar esta funcionalidad con el parámetro `core.autocrlf`. Si estás trabajando en una máquina Windows, ajustalo a `true`, para convertir finales LF en CRLF cuando extraigas código (checkout), puedes hacer esto:

```
$ git config --global core.autocrlf true
```

Si estás trabajando en una máquina Linux o Mac, entonces no te interesa convertir automáticamente los finales de línea al extraer código, sino que te interesa arreglar los posibles CRLF que pudieran aparecer accidentalmente. Puedes indicar a Git que convierta CRLF en LF al confirmar cambios (commit), pero no en el otro sentido; utilizando también el parámetro `core.autocrlf`:

```
$ git config --global core.autocrlf input
```

Este ajuste dejará los finales de línea CRLF en las extracciones de código (checkout), pero dejará los finales LF en sistemas Mac o Linux, y en el repositorio.

Si eres un programador Windows, trabajando en un entorno donde solo haya máquinas Windows, puedes desconectar esta funcionalidad, para almacenar CRLFs en el repositorio. Ajustando el parámetro a `false`:

```
$ git config --global core.autocrlf false
```

CORE.WHITESPACE

Git viene preajustado para detectar y resolver algunos de los problemas más típicos relacionados con los espacios en blanco. Puede vigilar acerca de seis tipos de problemas de espaciado: tres los tiene activados por defecto, pero se pueden desactivar; y tres vienen desactivados por defecto, pero se pueden activar.

Los que están activos de forma predeterminada son `blank-at-eol`, que busca espacios al final de la línea; `blank-at-eof`, que busca líneas en blanco al final del fichero y `space-before-tab`, que busca espacios delante de las tabulaciones al comienzo de una línea.

Los que están inactivos de forma predeterminada son `ident-with-non-tab`, que busca líneas que empiezan con espacios en blanco en lugar de tabulaciones (y se controla con la opción `tabwidth`); `tab-in-indent`, que busca tabulaciones en el trozo indentado de una línea; y `cr-at-eol`, que informa a Git de que los CR al final de las líneas son correctos.

Puedes decir a Git cuales de ellos deseas activar o desactivar, ajustando el parámetro `core.whitespace` con los valores on/off separados por comas. Puedes desactivarlos tanto dejandolos fuera de la cadena de ajustes, como añadiendo el prefijo - delante del valor. Por ejemplo, si deseas activar todos menos `cr-at-eol` puedes lanzar:

```
$ git config --global core.whitespace \
    trailing-space,space-before-tab,indent-with-non-tab
```

Git detectará posibles problemas cuando lance un comando `git diff`, e intentará destacarlos en otro color para que puedas corregirlos antes de confirmar cambios (`commit`). También pueden ser útiles estos ajustes cuando estás incorporando parches con `git apply`. Al incorporar parches, puedes pedirle a Git que te avise específicamente sobre determinados problemas de espaciado:

```
$ git apply --whitespace=warn <patch>
```

O puedes pedirle que intente corregir automáticamente los problemas antes de aplicar el parche:

```
$ git apply --whitespace=fix <patch>
```

Estas opciones se pueden aplicar también al comando `git rebase`. Si has confirmado cambios con problemas de espaciado, pero no los has enviado (push) aún “aguas arriba” (upstream). Puedes realizar una reorganización (rebase) con la opción `--whitespace=fix` para que Git corrija automáticamente los problemas según va reescribiendo los parches.

Configuración del servidor

No hay tantas opciones de configuración en el lado servidor de Git, pero hay unas pocas interesantes que merecen ser tenidas en cuenta.

RECEIVE.FSCKOBJECTS

Por defecto, Git no suele comprobar la consistencia de todos los objetos que recibe durante un envío (push), aunque Git tiene la capacidad para asegurarse de que cada objeto sigue casando con su suma de control SHA-1 y sigue apuntando a objetos válidos. No lo suele hacer en todos y cada uno de los envíos (push). Es una operación costosa, que, dependiendo del tamaño del repositorio, puede llegar a añadir mucho tiempo a cada operación de envío (push). De todas formas, si deseas que Git compruebe la consistencia de todos los objetos en todos los envíos, puedes forzarle a hacerlo ajustando a `true` el parámetro `receive.fsckObjects`:

```
$ git config --system receive.fsckObjects true
```

A partir de ese momento, Git comprobará la integridad del repositorio antes de aceptar ningún envío (push), para asegurarse de que no está introduciendo datos corruptos.

RECEIVE.DENYNONFASTFORWARDS

Si reorganizas (rebase) confirmaciones de cambio (commit) que ya habías enviado y tratas de enviarlas (push) de nuevo; o si intentas enviar una confirmación a una rama remota que no contiene la confirmación actualmente apuntada por la rama, normalmente, la operación te será denegada por la rama remota sobre la que pretendías realizarla. Habitualmente, este es el comportamiento más adecuado. Pero, en el caso de las reorganizaciones, cuando estás totalmente seguro de lo que haces, puedes forzar el envío, utilizando la opción `-f` en el comando `git push` a la rama remota.

Para impedir estos envíos forzados de referencias de avance no directo (no fast-forward) a ramas remotas, es para lo que se emplea el parámetro `receive.denyNonFastForwards`:

```
$ git config --system receive.denyNonFastForwards true
```

Otra manera de obtener el mismo resultado, es a través de los enganches (hooks) en el lado servidor, enganches de los que hablaremos en breve. Esta otra vía te permite realizar ajustes más finos, tales como denegar referencias de avance no directo, (non-fast-forwards), únicamente a un grupo de usuarios.

RECEIVE.DENYDELETEST

Uno de los cortocircuitos que suelen utilizar los usuarios para saltarse la política de `denyNonFastForwards`, suele ser el borrar la rama y luego volver a enviarla de vuelta con la nueva referencia. Puedes evitar esto poniendo a `true` el parámetro `receive.denyDeletes`:

```
$ git config --system receive.denyDeletes true
```

Esto impide el borrado de ramas o de etiquetas por medio de un envío a través de la mesa (push across the board), --ningún usuario lo podrá hacer--. Para borrar ramas remotas, tendrás que borrar los archivos de referencia manualmente sobre el propio servidor. Existen también algunas otras maneras más interesantes de hacer esto mismo, pero para usuarios concretos, a través de permisos (ACLs); tal y como veremos en “[Un ejemplo de implantación de una determinada política en Git](#)”.

Git Attributes

Algunos de los ajustes que hemos visto, pueden ser especificados para un camino (path) concreto, de tal forma que Git los aplicará únicamente para una carpeta o para un grupo de archivos determinado. Estos ajustes específicos relacionados con un camino, se denominan atributos en Git. Y se pueden fijar, bien mediante un archivo `.gitattribute` en uno de los directorios de tu proyecto (normalmente en la raíz del proyecto), o bien mediante el archivo `git/info/attributes` en el caso de no querer guardar el archivo de atributos dentro de tu proyecto.

Por medio de los atributos, puedes hacer cosas tales como indicar diferentes estrategias de fusión para archivos o carpetas concretas de tu proyecto, decirle a Git cómo comparar archivos no textuales, o indicar a Git que filtre ciertos contenidos antes de guardarlos o de extraerlos del repositorio Git. En esta sección, aprenderás acerca de algunos atributos que puedes asignar a ciertos caminos (paths) dentro de tu proyecto Git, viendo algunos ejemplos de cómo utilizar sus funcionalidades de manera práctica.

Archivos binarios

Un buen truco donde utilizar los atributos Git es para indicarle cuáles de los archivos son binarios (en los casos en que Git no podría llegar a determinarlo por sí mismo), dándole a Git instrucciones especiales sobre cómo tratar estos archivos. Por ejemplo, algunos archivos de texto se generan automáticamente y no tiene sentido compararlos; mientras que algunos archivos binarios sí que pueden ser comparados. Vamos a ver cómo indicar a Git cuál es cuál.

IDENTIFICACIÓN DE ARCHIVOS BINARIOS

Algunos archivos aparentan ser textuales, pero a efectos prácticos merece más la pena tratarlos como binarios. Por ejemplo, los proyectos Xcode en un Mac contienen un archivo terminado en `.pbxproj`. Este archivo es básicamente una base de datos JSON (datos Javascript en formato de texto plano), escrita directamente por el IDE para almacenar aspectos tales como tus ajustes de compilación. Aunque técnicamente es un archivo de texto (ya que su contenido lo forman caracteres UTF-8). Realmente nunca lo tratarás como tal, porque en realidad es una base de datos ligera (y no puedes fusionar sus contenidos si dos personas lo cambian, porque las comparaciones no son de utilidad). Éstos son archivos destinados a ser tratados de forma automatizada. Y es preferible tratarlos como si fueran archivos binarios.

Para indicar a Git que trate todos los archivos `pbxproj` como binarios, puedes añadir esta línea a tu archivo `.gitattributes`:

```
*.pbxproj binary
```

A partir de ahora, Git no intentará convertir ni corregir problemas CRLF en los finales de línea; ni intentará hacer comparaciones ni mostar diferencias de este archivo cuando lances comandos `git show` o `git diff` en tu proyecto.

COMPARACIÓN ENTRE ARCHIVOS BINARIOS

Puedes utilizar los atributos Git para comparar archivos binarios. Se consigue diciéndole a Git la forma de convertir los datos binarios en texto, consiguiendo así que puedan ser comparados con la herramienta habitual de comparación textual.

En primer lugar, utilizarás esta técnica para resolver uno de los problemas más engorrosos conocidos por la humanidad: el control de versiones en documentos Word. Todo el mundo conoce el hecho de que Word es el editor más horroroso de cuantos hay; pero, desgraciadamente, todo el mundo lo usa. Si deseas controlar versiones en documentos Word, puedes añadirlos a un repositorio Git e ir realizando confirmaciones de cambio (commit) cada vez. Pero, ¿qué ganas con ello?. Si lanzas un comando `git diff`, lo único que verás será algo tal como:

```
$ git diff
diff --git a/chapter1.docx b/chapter1.docx
index 88839c4..4afcb7c 100644
Binary files a/chapter1.docx and b/chapter1.docx differ
```

No puedes comparar directamente dos versiones, a no ser que extraigas ambas y las compares manualmente, ¿no?. Pero resulta que puedes hacerlo bastante mejor utilizando los atributos Git. Poniendo lo siguiente en tu archivo `.gitattributes`:

```
*.docx diff=word
```

Así decimos a Git que sobre cualquier archivo coincidente con el patrón indicado, (.docx), ha de utilizar el filtro “word” cuando intente hacer una comparación con él. ¿Qué es el filtro “word”? Tienes que configurarlo tú mismo. Por ejemplo, puedes configurar Git para que utilice el programa `docx2txt` para convertir los documentos Word en archivos de texto planos, archivos sobre los que poder realizar comparaciones sin problemas.

En primer lugar, necesitas instalar `docx2txt`, obteniéndolo de: <http://docx2txt.sourceforge.net>. Sigue las instrucciones del fichero `INSTALL` e instálalo en algún sitio donde se pueda ejecutar desde la shell. A continuación, escribe un script que sirva para convertir el texto al formato que Git necesita, utilizando `docx2txt`:

```
#!/bin/bash
docx2txt.pl $1 -
```

No olvides poner los permisos de ejecución al script (`chmod a+x`). Finalmente, configura Git para usar el script:

```
$ git config diff.word.textconv docx2txt
```

Ahora Git ya sabrá que si intentas comparar dos archivos, y cualquiera de ellos finaliza en `.docx`, lo hará a través del filtro “word”, que se define con el programa `docx2txt`. Esto provoca la creación de versiones de texto de los ficheros Word antes de intentar compararlos.

Por ejemplo, el capítulo 1 de este libro se convirtió a Word y se envió al repositorio Git. Cuando añadimos posteriormente un nuevo párrafo, el `git diff` muestra lo siguiente:

```
$ git diff
diff --git a/chapter1.docx b/chapter1.docx
index 0b013ca..ba25db5 100644
--- a/chapter1.docx
+++ b/chapter1.docx
@@ -2,6 +2,7 @@
    This chapter will be about getting started with Git. We will begin at the beginning by exploring
    1.1. About Version Control
    What is "version control", and why should you care? Version control is a system that records changes in files over time.
+Testing: 1, 2, 3.
    If you are a graphic or web designer and want to keep every version of an image or layout (per
    1.1.1. Local Version Control Systems
    Many people's version-control method of choice is to copy files into another directory (per
```

Git nos muestra que se añadió la línea “Testing: 1, 2, 3.”, lo cual es correcto. No es perfecto (los cambios de formato, por ejemplo, no los mostrará) pero sirve en la mayoría de los casos.

Otro problema donde puede ser útil esta técnica, es en la comparación de imágenes. Un camino puede ser pasar los archivos JPEG a través de un filtro para extraer su información EXIF (los metadatos que se graban dentro de la mayoría de formatos gráficos). Si te descargas e instalas el programa `exiftool`, puedes utilizarlo para convertir tus imágenes a textos (metadatos), de tal forma que `diff` podrá al menos mostrarte algo útil de cualquier cambio que se produzca:

```
$ echo '*.*.png diff=exif' >> .gitattributes
$ git config diff.exif.textconv exiftool
```

Si reemplazas cualquier imagen en el proyecto y ejecutas `git diff`, verás algo como:

```
diff --git a/image.png b/image.png
index 88839c4..4afcb7c 100644
--- a/image.png
+++ b/image.png
@@ -1,12 +1,12 @@
    ExifTool Version Number      : 7.74
    -File Size                 : 70 kB
    -File Modification Date/Time: 2009:04:21 07:02:45-07:00
    +File Size                 : 94 kB
    +File Modification Date/Time: 2009:04:21 07:02:43-07:00
    File Type                  : PNG
    MIME Type                  : image/png
    -Image Width               : 1058
    -Image Height              : 889
    +Image Width               : 1056
    +Image Height              : 827
    Bit Depth                  : 8
    Color Type                 : RGB with Alpha
```

Aquí se ve claramente que ha cambiado el tamaño del archivo y las dimensiones de la imagen.

Expansión de palabras clave

Algunos usuarios de sistemas SVN o CVS, echan de menos el disponer de expansiones de palabras clave al estilo de las que dichos sistemas tienen. El principal problema para hacerlo en Git reside en la imposibilidad de modificar los ficheros con información relativa a la confirmación de cambios (commit). Debido a que Git calcula sus sumas de comprobación antes de las confirmaciones. De todas formas, es posible inyectar textos en un archivo cuando lo extraemos del repositorio (checkout) y quitarlos de nuevo antes de devolverlo al repositorio (commit). Los atributos Git admiten dos maneras de realizarlo.

La primera, es inyectando automáticamente la suma de comprobación SHA-1 de un gran objeto binario (blob) en un campo `Id` dentro del archivo. Si colocas este atributo en un archivo o conjunto de archivos, Git lo sustituirá por la suma de comprobación SHA-1 la próxima vez que lo/s extraiga/s. Es importante destacar que no se trata de la suma SHA de la confirmación de cambios (commit), sino del propio objeto binario (blob):

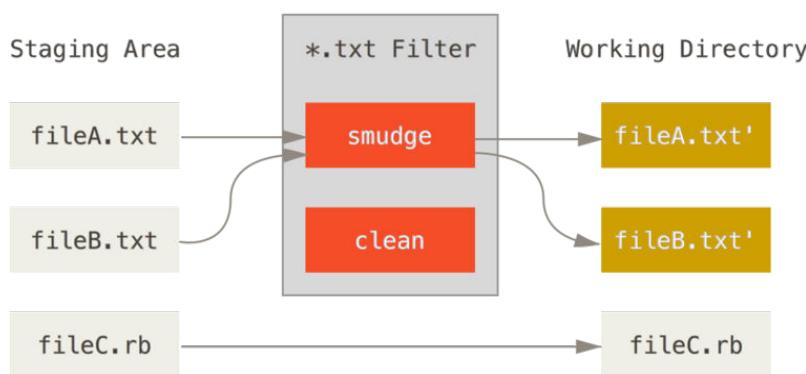
```
$ echo '*.txt ident' >> .gitattributes
$ echo '$Id$' > test.txt
```

La próxima vez que extraigas el archivo, Git le habrá inyectado el SHA-1 del objeto binario (blob):

```
$ rm test.txt
$ git checkout -- test.txt
$ cat test.txt
$Id: 42812b7653c7b88933f8a9d6cad0ca16714b9bb3 $
```

Pero esto tiene un uso bastante limitado. Si has utilizado alguna vez las sustituciones de CVS o de Subversion, sabrás que pueden incluir una marca de fecha (la suma de comprobación SHA no es igual de util, ya que, por ser bastante aleatoria, es imposible deducir si una suma SHA es anterior o posterior a otra).

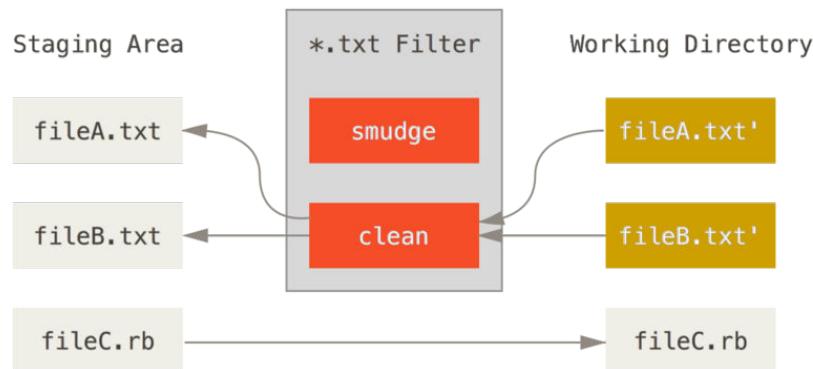
Aunque resulta que también puedes escribir tus propios filtros para realizar sustituciones en los archivos al guardar o recuperar (commit/checkout). Se trata de los filtros “clean” y “smudge”. En el archivo ‘.gitattributes’ puedes indicar filtros para carpetas o archivos determinados y luego preparar tus propios scripts para procesarlos justo antes de confirmar cambios en ellos (“clean”, ver **Figure 8-2**), o justo antes de recuperarlos (“smudge”, ver **Figure 8-3**). Estos filtros pueden utilizarse para realizar todo tipo de acciones útiles.

**FIGURE 8-2**

Ejecución de filtro “smudge” en el checkout.

FIGURE 8-3

Ejecución de filtro “clean” antes de confirmar el cambio.



El mensaje de confirmación para esta funcionalidad nos da un ejemplo simple: el de pasar todo tu código fuente C por el programa `indent` antes de almacenarlo. Puedes hacerlo poniendo los atributos adecuados en tu archivo `.gitattributes`, para filtrar los archivos `*.c` a través de “`indent`”:

```
*.c filter=indent
```

E indicando después que el filtro “`indent`” actuará al manchar (smudge) y al limpiar (clean):

```
$ git config --global filter.indent.clean indent
$ git config --global filter.indent.smudge cat
```

En este ejemplo, cuando confirmes cambios (commit) en archivos con extensión `*.c`, Git los pasará previamente a través del programa `indent` antes de confirmarlos, y los pasará a través del programa `cat` antes de extraerlos de vuelta al disco. El programa `cat` es básicamente transparente: de él salen los mismos datos que entran. El efecto final de esta combinación es el de filtrar todo el código fuente C a través de `indent` antes de confirmar cambios en él.

Otro ejemplo interesante es el de poder conseguir una expansión de la clave `$Date$` del estilo de RCS. Para hacerlo, necesitas un pequeño script que coja el nombre de un archivo, localice la fecha de la última confirmación de cambios en el proyecto, e inserte dicha información en el archivo. Este podría ser un pequeño script Ruby para hacerlo:

```
#!/usr/bin/env ruby
data = STDIN.read
```

```
last_date = `git log --pretty=format:"%ad" -1`
puts data.gsub('$Date$', '$Date: ' + last_date.to_s + '$')
```

Simplemente, utiliza el comando `git log` para obtener la fecha de la última confirmación de cambios, y sustituye con ella todas las cadenas `$Date$` que encuentre en el flujo de entrada `stdin`; imprimiendo luego los resultados. Debería de ser sencillo de implementarlo en cualquier otro lenguaje que domines.

Puedes llamar `expanddate` a este archivo y ponerlo en el path de ejecución. Tras ello, has de poner un filtro en Git (podemos llamarle `dater`), e indicarle que use el filtro `expanddate` para manchar (smudge) los archivos al extraerlos (`checkout`). Puedes utilizar una expresión Perl para limpiarlos (`clean`) al almacenarlos (`commit`):

```
$ git config filter.dater.smudge expand_date
$ git config filter.dater.clean 'perl -pe "s/\$\$Date[^\\\$]*\\\$/\$\$Date\\\$/"'
```

Esta expresión Perl extrae cualquier cosa que vea dentro de una cadena `$Date$`, para devolverla a como era en un principio. Una vez preparado el filtro, puedes comprobar su funcionamiento preparando un archivo que contenga la clave `$Date$` e indicando a Git cual es el atributo para reconocer ese tipo de archivo:

```
$ echo '# $Date$' > date_test.txt
$ echo 'date*.txt filter=dater' >> .gitattributes
```

Al confirmar cambios (`commit`) y luego extraer (`checkout`) el archivo de vuelta, verás la clave sustituida:

```
$ git add date_test.txt .gitattributes
$ git commit -m "Testing date expansion in Git"
$ rm date_test.txt
$ git checkout date_test.txt
$ cat date_test.txt
# $Date: Tue Apr 21 07:26:52 2009 -0700$
```

Esta es una muestra de lo poderosa que puede resultar esta técnica para aplicaciones personalizadas. No obstante, debes de ser cuidadoso, ya que el archivo `.gitattributes` se almacena y se transmite junto con el proyecto; pero no así el propio filtro, (en este caso, `dater`), sin el cual no puede funcionar.

Cuando diseñas este tipo de filtros, han de estar pensados para que el proyecto continue funcionando correctamente incluso cuando fallen.

Exportación del repositorio

Los atributos de Git permiten realizar algunas cosas interesantes cuando exportas un archivo de tu proyecto.

EXPORT-IGNORE

Puedes indicar a Git que ignore y no exporte ciertos archivos o carpetas cuando genera un archivo de almacenamiento. Cuando tienes alguna carpeta o archivo que no deseas incluir en tus registros, pero quieras tener controlado en tu proyecto, puedes marcarlos a través del atributo `export-ignore`.

Por ejemplo, supongamos que tienes algunos archivos de pruebas en la carpeta `test/`, y que no tiene sentido incluirlos en los archivos comprimidos (tarball) al exportar tu proyecto. Puedes añadir la siguiente línea al archivo de atributos de Git:

```
test/ export-ignore
```

A partir de ese momento, cada vez que lances el comando `git archive` para crear un archivo comprimido de tu proyecto, esa carpeta no se incluirá en él.

EXPORT-SUBST

Otra cosa que puedes realizar sobre tus archivos es algún tipo de sustitución simple de claves. Git te permite poner la cadena `$Format:$` en cualquier archivo, con cualquiera de las claves de formateo de `--pretty=format` que vimos en el capítulo 2. Por ejemplo, si deseas incluir un archivo llamado `LAST COMMIT` en tu proyecto, y poner en él automáticamente la fecha de la última confirmación de cambios cada vez que lances el comando `git archive`:

```
$ echo 'Last commit date: $Format:%cd$' > LAST_COMMIT
$ echo "LAST_COMMIT export-subst" >> .gitattributes
$ git add LAST_COMMIT .gitattributes
$ git commit -am 'adding LAST_COMMIT file for archives'
```

Cuando lances la orden `git archive`, lo que la gente verá en ese archivo cuando lo abra será:

```
$ cat LAST_COMMIT
Last commit date: $Format:Tue Apr 21 08:38:48 2009 -0700$
```

Estrategias de fusión (merge)

También puedes utilizar los atributos Git para indicar distintas estrategias de fusión para archivos específicos de tu proyecto. Una opción muy útil es la que nos permite indicar a Git que no intente fusionar ciertos archivos concretos cuando tengan conflictos, manteniendo en su lugar tus archivos sobre los de cualquier otro.

Puede ser interesante si una rama de tu proyecto es divergente o está especializada, pero deseas seguir siendo capaz de fusionar cambios de vuelta desde ella, e ignorar ciertos archivos. Digamos que tienes un archivo de datos denominado `database.xml`, distinto en las dos ramas, y que deseas fusionar en la otra rama sin perturbarlo. Puedes ajustar un atributo tal como:

```
database.xml merge=ours
```

Y luego definir una estrategia `ours` con:

```
$ git config --global merge.ours.driver true
```

Si fusionas en la otra rama, en lugar de tener conflictos de fusión con el fichero `database.xml`, verás algo como:

```
$ git merge topic
Auto-merging database.xml
Merge made by recursive.
```

Y el archivo `database.xml` permanecerá inalterado en cualquier que fuera la versión que tenías originalmente.

Puntos de enganche en Git

Al igual que en otros sistemas de control de versiones, Git también cuenta con mecanismos para lanzar scripts de usuario cuando suceden ciertas acciones importantes, llamados puntos de enganche (hooks). Hay dos grupos de esos puntos de lanzamiento: los del lado cliente y los del lado servidor. Los puntos del lado cliente están relacionados con operaciones tales como la confirmación de

cambios (commit) o la fusión (merge). Los del lado servidor están relacionados con operaciones tales como la recepción de contenidos enviados (push) a un servidor. Estos puntos de enganche pueden utilizarse para multitud de aplicaciones. Vamos a ver unas pocas de ellas.

Instalación de un punto de enganche

Los puntos de enganche se guardan en la subcarpeta `hooks` de la carpeta Git. En la mayoría de proyectos, estará en `.git/hooks`. Por defecto, esta carpeta contiene unos cuantos scripts de ejemplo. Algunos de ellos son útiles por sí mismos; pero su misión principal es la de documentar las variables de entrada para cada script. Todos los ejemplos se han escrito como scripts de shell, con algo de código Perl embebido en ellos. Pero cualquier tipo de script ejecutable que tenga el nombre adecuado puede servir igual de bien. Los puedes escribir en Ruby o en Python o en cualquier lenguaje de scripting con el que trabajes. Si quieres usar los ejemplos que trae Git, tendrás que renombrarlos, ya que los ejemplos acaban su nombre en `.sample`.

Para activar un punto de enganche para un script, pon el archivo correspondiente en la carpeta `hooks`; con el nombre adecuado y con la marca de ejecutable. A partir de ese momento, será automáticamente lanzado cuando se dé la acción correspondiente. Vamos a ver la mayoría de nombres de puntos de enganche disponibles.

Puntos de enganche del lado cliente

Hay muchos de ellos. En esta sección los dividiremos en puntos de enganche en el flujo de trabajo de confirmación de cambios, puntos en el flujo de trabajo de correo electrónico y todos los demás.

Observa que los puntos de enganche del lado del cliente **no se copian** cuando clonas el repositorio. Si quieres que tengan un efecto para forzar una política es necesario que esté en el lado del cliente. Por ejemplo, mira en “[Un ejemplo de implantación de una determinada política en Git](#)”.

PUNTOS EN EL FLUJO DE TRABAJO DE CONFIRMACIÓN DE CAMBIOS

Los primeros cuatro puntos de enganche están relacionados con el proceso de confirmación de cambios.

Primero se activa el punto de enganche `pre-commit`, incluso antes de que teclees el mensaje de confirmación. Se suele utilizar para inspeccionar la instantánea (snapshot) que vas a confirmar, para ver si has olvidado algo, para

asegurar que las pruebas se ejecutan, o para revisar cualquier aspecto que necesites inspeccionar en el código. Saliendo con un valor de retorno distinto de cero, se aborta la confirmación de cambios. Aunque siempre puedes saltartelo con la orden `git commit --no-verify`. Puede ser útil para realizar tareas tales como revisar el estilo del código (lanzando `lint` o algo equivalente), revisar los espacios en blanco de relleno (el script de ejemplo hace exactamente eso), o revisar si todos los nuevos métodos llevan la adecuada documentación.

El punto de enganche `prepare-commit-msg` se activa antes de arrancar el editor del mensaje de confirmación de cambios, pero después de crearse el mensaje por defecto. Te permite editar el mensaje por defecto, antes de que lo vea el autor de la confirmación de cambios. Este punto de enganche recibe varias entradas: la ubicación (path) del archivo temporal donde se almacena el mensaje de confirmación, el tipo de confirmación y la clave SHA-1 si estamos enmendando un `commit` existente. Este punto de enganche no tiene mucha utilidad para las confirmaciones de cambios normales; pero sí para las confirmaciones donde el mensaje por defecto es autogenerado, como en las confirmaciones de fusiones (`merge`), los mensajes con plantilla, las confirmaciones aplastadas (`squash`), o las confirmaciones de corrección (`amend`). Se puede utilizar combinándolo con una plantilla de confirmación, para poder insertar información automáticamente.

El punto de enganche `commit-msg` recibe un parámetro: la ubicación (path) del archivo temporal que contiene el mensaje de confirmación actual. Si este script termina con un código de salida distinto de cero, Git aborta el proceso de confirmación de cambios; permitiendo así validar el estado del proyecto o el mensaje de confirmación antes de permitir continuar. En la última parte de este capítulo, veremos cómo podemos utilizar este punto de enganche para revisar si el mensaje de confirmación es conforme a un determinado patrón obligatorio.

Después de completar todo el proceso de confirmación de cambios, es cuando se lanza el punto de enganche `post-commit`. Este no recibe ningún parámetro, pero podemos obtener fácilmente la última confirmación de cambios con el comando `git log -1 HEAD`. Habitualmente, este script final se suele utilizar para realizar notificaciones o tareas similares.

PUNTOS EN EL FLUJO DE TRABAJO DEL CORREO ELECTRÓNICO

Tienes disponibles tres puntos de enganche en el lado cliente para interactuar con el flujo de trabajo de correo electrónico. Todos ellos se invocan al utilizar el comando `git am`, por lo que si no utilizas dicho comando, puedes saltar directamente a la siguiente sección. Si recibes parches a través de `correo-e` preparados con `git format-patch`, es posible que parte de lo descrito en esta sección te pueda ser útil.

El primer punto de enganche que se activa es `applypatch-msg`. Recibe un solo argumento: el nombre del archivo temporal que contiene el mensaje de confirmación propuesto. Git abortará la aplicación del parche si este script termina con un código de salida distinto de cero. Puedes utilizarlo para asegurarte de que el mensaje de confirmación esté correctamente formateado o para normalizar el mensaje permitiendo al script que lo edite sobre la marcha.

El siguiente punto de enganche que se activa al aplicar parches con `git am` es el punto `pre-applypatch`. No recibe ningún argumento de entrada y se lanza después de que el parche haya sido aplicado, por lo que puedes utilizarlo para revisar la situación (snapshot) antes de confirmarla. Con este script, puedes lanzar pruebas o similares para chequear el árbol de trabajo. Si falta algo o si alguna de las pruebas falla, saliendo con un código de salida distinto de cero abortará el comando `git am` sin confirmar el parche.

El último punto de enganche que se activa durante una operación `git am` es el punto `post-applypatch`. Puedes utilizarlo para notificar de su aplicación al grupo o al autor del parche. No puedes detener el proceso de parcheo con este script.

OTROS PUNTOS DE ENGANCHE DEL LADO CLIENTE

El punto `pre-rebase` se activa antes de cualquier reorganización y puede abortarla si retorna con un código de salida distinto de cero. Puedes usarlo para impedir reorganizaciones de cualquier confirmación de cambios ya enviada (push) a algún servidor. El script de ejemplo para `pre-rebase` hace precisamente eso, aunque asumiendo que `next` es el nombre de la rama publicada. Si lo vas a utilizar, tendrás que modificarlo para que se ajuste al nombre que tenga tu rama publicada.

El punto de enganche `post-rewrite` se ejecuta con los comandos que reemplazan confirmaciones de cambio, como `git commit --amend` y `git rebase` (pero no con `git filter-branch`). Su único argumento es el comando que disparará la reescritura, y recibe una lista de reescrituras por la entrada estándar (`stdin`). Este enganche tiene muchos usos similares a los puntos `post-checkout` y `post-merge`.

Tras completarse la ejecución de un comando `git checkout`, es cuando se activa el punto de enganche `post-checkout`. Lo puedes utilizar para ajustar tu carpeta de trabajo al entorno de tu proyecto. Entre otras cosas, puedes mover grandes archivos binarios de los que no quieras llevar control, puedes autogenerar documentación, y otras cosas.

El punto de enganche `post-merge` se activa tras completarse la ejecución de un comando `git merge`. Puedes utilizarlo para recuperar datos de tu carpeta de trabajo que Git no puede controlar, como por ejemplo datos relativos a

permisos. Este punto de enganche puede utilizarse también para comprobar la presencia de ciertos archivos, externos al control de Git, que deseas copiar cada vez que cambie la carpeta de trabajo.

El punto `pre-push` se ejecuta durante un `git push`, justo cuando las referencias remotas se han actualizado, pero antes de que los objetos se transfieran. Recibe como parámetros el nombre y la localización del remoto, y una lista de referencias para ser actualizadas, a través de la entrada estándar. Puedes utilizarlo para validar un conjunto de actualizaciones de referencias antes de que la operación de push tenga lugar (ya que si el script retorna un valor distinto de cero, se abortará la operación).

En ocasiones, Git realizará una recolección de basura como parte de su funcionamiento habitual, llamando a `git gc --auto`. El punto de enganche `pre-auto-gc` es el que se llama justo antes de realizar dicha recolección de basura, y puede utilizarse para notificarte que tiene lugar dicha operación, o para poderla abortar si se considera que no es un buen momento.

Puntos de enganche del lado servidor

Aparte de los puntos del lado cliente, como administrador de sistemas, puedes utilizar un par de puntos de enganche importantes en el lado servidor; para implementar prácticamente cualquier tipo de política que quieras mantener en tu proyecto. Estos scripts se lanzan antes y después de cada envío (push) al servidor. El script previo, puede terminar con un código de salida distinto de cero y abortar el envío, devolviendo el correspondiente mensaje de error al cliente. Este script puede implementar políticas de recepción tan complejas como deseas.

PRE-RECEIVE

El primer script que se activa al manejar un envío de un cliente es el correspondiente al punto de enganche `pre-receive`. Recibe una lista de referencias que se están enviando (push) desde la entrada estándar (`stdin`); y, si termina con un código de salida distinto de cero, ninguna de ellas será aceptada. Puedes utilizar este punto de enganche para realizar tareas tales como la de comprobar que ninguna de las referencias actualizadas no son de avance directo (`non-fast-forward`); o para comprobar que el usuario que realiza el envío tiene realmente permisos para crear, borrar o modificar cualquiera de los archivos que está tratando de cambiar.

UPDATE

El punto de enganche `update` es muy similar a `pre-receive`, pero con la diferencia de que se activa una vez por cada rama que se está intentando actualizar con el envío. Si la persona que realiza el envío intenta actualizar varias ramas, `pre-receive` se ejecuta una sola vez, mientras que `update` se ejecuta tantas veces como ramas se estén actualizando. El lugar de recibir datos desde la entrada estándar (`stdin`), este script recibe tres argumentos: el nombre de la rama, la clave SHA-1 a la que esta apuntada antes del envío, y la clave SHA-1 que el usuario está intentando enviar. Si el script `update` termina con un código de salida distinto de cero, únicamente los cambios de esa rama son rechazados; el resto de ramas continuarán con sus actualizaciones.

POST-RECEIVE

El punto de enganche `post-receive` se activa cuando termina todo el proceso, y se puede utilizar para actualizar otros servicios o para enviar notificaciones a otros usuarios. Recibe los mismos datos que `pre-receive` desde la entrada estándar. Algunos ejemplos de posibles aplicaciones pueden ser la de alimentar una lista de correo-e, avisar a un servidor de integración continua, o actualizar un sistema de seguimiento de tickets de servicio (pudiendo incluso procesar el mensaje de confirmación para ver si hemos de abrir, modificar o dar por cerrado algún ticket). Este script no puede detener el proceso de envío, pero el cliente no se desconecta hasta que no se completa su ejecución; por tanto, has de ser cuidadoso cuando intentes realizar con él tareas que puedan requerir mucho tiempo.

Un ejemplo de implantación de una determinada política en Git

En esta sección, utilizarás lo aprendido para establecer un flujo de trabajo en Git que: compruebe si los mensajes de confirmación de cambios encajan en un determinado formato, obligue a realizar solo envíos de avance directo, y permita solo a ciertos usuarios modificar ciertas carpetas del proyecto. Para ello, has de preparar los correspondientes scripts de cliente (para ayudar a los desarrolladores a saber de antemano si sus envíos van a ser rechazados o no), y los correspondientes scripts de servidor (para obligar a cumplir esas políticas).

He usado Ruby para escribir los ejemplos, tanto porque es mi lenguaje preferido de scripting y porque creo que es el más parecido a pseudocódigo; de tal forma que puedes ser capaz de seguir el código, incluso si no conoces Ruby. Pero, puede ser igualmente válido cualquier otro lenguaje. Todos los script de

ejemplo que vienen de serie con Git están escritos en Perl o en Bash shell, por lo que tienes bastantes ejemplos en esos lenguajes de scripting.

Punto de enganche en el lado servidor

Todo el trabajo del lado servidor va en el script update de la carpeta hooks. Dicho script se lanza cada vez que alguien sube algo a alguna rama, y tiene tres argumentos:

- El nombre de la referencia que se está subiendo
- La vieja revisión de la rama que se está modificando
- La nueva revisión que se está subiendo a la rama

También puedes tener acceso al usuario que está enviando, si este los envia a través de SSH. Si has permitido a cualquiera conectarse con un mismo usuario (como “git”, por ejemplo), has tenido que dar a dicho usuario una envoltura (shell wrapper) que te permite determinar cuál es el usuario que se conecta según sea su clave pública, permitiéndote fijar una variable de entorno especificando dicho usuario. Aquí, asumiremos que el usuario conectado queda reflejado en la variable de entorno \$USER, de tal forma que el script update comienza recogiendo toda la información que necesitas:

```
#!/usr/bin/env ruby

$refname = ARGV[0]
$oldrev = ARGV[1]
$newrev = ARGV[2]
$user = ENV['USER']

puts "Enforcing Policies..."
puts "({$refname}) ({$oldrev[0..6]}) ({$newrev[0..6]})"
```

Sí, estoy usando variables globales. No me juzgues por ello, que es más sencillo mostrarlo de esta manera.

OBLIGANDO A UTILIZAR UN FORMATO ESPECÍFICO EN EL MENSAJE DE COMMIT

Tu primer desafío es asegurarte que todos y cada uno de los mensajes de confirmación de cambios se ajustan a un determinado formato. Simplemente, por fijar algo concreto, supongamos que cada mensaje ha de incluir un texto tal como “ref: 1234”, porque quieras enlazar cada confirmación de cambios con una determinada entrada de trabajo en un sistema de control. Has de mirar en cada confirmación de cambios (commit) recibida, para ver si contiene ese texto; y, si

no lo trae, salir con un código distinto de cero, de tal forma que el envío (push) sea rechazado.

Puedes obtener la lista de las claves SHA-1 de todos los confirmaciones de cambios enviadas cogiendo los valores de \$newrev y de \$oldrev, y pasándolos a comando de mantenimiento de Git llamado `git rev-list`. Este comando es básicamente el mismo que `git log`, pero por defecto, imprime solo los valores SHA-1 y nada más. Con él, puedes obtener la lista de todas las claves SHA que se han introducido entre una clave SHA y otra clave SHA dadas; obtendrás algo así como esto:

```
$ git rev-list 538c33..d14fc7
d14fc7c847ab946ec39590d87783c69b031bdfb7
9f585da4401b0a3999e84113824d15245c13f0be
234071a1be950e2a8d078e6141f5cd20c1e61ad3
dfa04c9ef3d5197182f13fb5b9b1fb7717d2222a
17716ec0f1ff5c77eff40b7fe912f9f6cf0e475
```

Puedes coger esta salida, establecer un bucle para recorrer cada una de esas confirmaciones de cambios, coger el mensaje de cada una y comprobarlo contra una expresión regular de búsqueda del patrón deseado.

Tienes que imaginarte cómo puedes obtener el mensaje de cada una de esas confirmaciones de cambios a comprobar. Para obtener los datos “en crudo” de una confirmación de cambios, puedes utilizar otro comando de mantenimiento de Git denominado `git cat-file`. En **Chapter 10** volveremos en detalle sobre estos comandos de mantenimiento; pero, por ahora, esto es lo que obtienes con dicho comando:

```
$ git cat-file commit ca82a6
tree cfda3bf379e4f8dba8717dee55aab78aef7f4daf
parent 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
author Scott Chacon <schacon@gmail.com> 1205815931 -0700
committer Scott Chacon <schacon@gmail.com> 1240030591 -0700

changed the version number
```

Una vía sencilla para obtener el mensaje, es la de ir hasta la primera línea en blanco y luego coger todo lo que siga a esta. En los sistemas Unix, lo puedes realizar con el comando `sed`:

```
$ git cat-file commit ca82a6 | sed '1,/^\$/d'
changed the version number
```

Puedes usar este “truco de magia” para coger el mensaje de cada confirmación de cambios que se está enviando y salir si localizas algo que no cuadra en alguno de ellos. Para salir del script y rechazar el envío, recuerda que debes salir con un código distinto de cero. El método completo será algo así como:

```
$regex = /\[ref: (\d+)\]/

# enforced custom commit message format
def check_message_format
  missed_revs = `git rev-list ${oldrev}..${newrev}`.split("\n")
  missed_revs.each do |rev|
    message = `git cat-file commit ${rev} | sed '1,/^\$/d'`
    if !$regex.match(message)
      puts "[POLICY] Your message is not formatted correctly"
      exit 1
    end
  end
end
check_message_format
```

Poniendo esto en tu script `update`, serán rechazadas todas las actualizaciones que contengan cambios con mensajes que no se ajusten a tus reglas.

IMPLEMENTANDO UN SISTEMA DE LISTAS DE CONTROL DE ACCESO (ACL)

Imaginemos que deseas implementar un sistema de control de accesos (Access Control List, ACL), para vigilar qué usuarios pueden enviar (push) cambios a qué partes de tus proyectos. Algunas personas tendrán acceso completo, y otras tan solo acceso a ciertas carpetas o a ciertos archivos. Para implementar esto, has de escribir esas reglas de acceso en un archivo denominado `acl` ubicado en tu repositorio git básico (bare) en el servidor. Y tienes que preparar el enganche `update` para hacerle consultar esas reglas, mirar los archivos que están siendo subidos en las confirmaciones de cambio (commit) enviadas (push), y determinar así si el usuario emisor del envío tiene o no permiso para actualizar esos archivos.

El primer paso es escribir tu lista de control de accesos (ACL). Su formato es muy parecido al del mecanismo ACL de CVS: utiliza una serie de líneas donde el primer campo es `avail` o `unavail` (permitido o no permitido), el segundo campo es una lista de usuarios separados por comas, y el último campo es la ubicación (path) sobre el que aplicar la regla (dejarlo en blanco equivale a un acceso abierto). Cada uno de esos campos se separan entre sí con el carácter barra vertical (|).

Por ejemplo, si tienes un par de administradores, algunos redactores técnicos con acceso a la carpeta doc, y un desarrollador que únicamente accede a las carpetas lib y tests, el archivo ACL resultante sería:

```
avail|nickh,pjhyett,defunkt,tpw
avail|usinclair,cdickens,ebronte|doc
avail|schacon|lib
avail|schacon|tests
```

Para implementarlo, hemos de leer previamente estos datos en una estructura que podamos emplear. En este caso, por razones de simplicidad, vamos a mostrar únicamente la forma de implementar las directivas avail (permitir). Este es un método que te devuelve un array asociativo cuya clave es el nombre del usuario y su valor es un array de ubicaciones (paths) donde ese usuario tiene acceso de escritura:

```
def get_acl_access_data(acl_file)
  # read in ACL data
  acl_file = File.read(acl_file).split("\n").reject { |line| line == '' }
  access = {}
  acl_file.each do |line|
    avail, users, path = line.split(' ')
    next unless avail == 'avail'
    users.split(',').each do |user|
      access[user] ||= []
      access[user] << path
    end
  end
  access
end
```

Si lo aplicamos sobre la lista ACL descrita anteriormente, este método get_acl_access_data devolverá una estructura de datos similar a esta:

```
{"defunkt"=>[nil],
 "tpw"=>[nil],
 "nickh"=>[nil],
 "pjhyett"=>[nil],
 "schacon"=>["lib", "tests"],
 "cdickens"=>["doc"],
 "usinclair"=>["doc"],
 "ebronte"=>["doc"]}
```

Una vez tienes los permisos en orden, necesitas averiguar las ubicaciones modificadas por las confirmaciones de cambios enviadas; de tal forma que

puedas asegurarte de que el usuario que las está enviando tiene realmente permiso para modificarlas.

Puedes comprobar fácilmente qué archivos han sido modificados en cada confirmación de cambios, utilizando la opción `--name-only` del comando `git log` (citado brevemente en el capítulo 2):

```
$ git log -1 --name-only --pretty=format:'' 9f585d
README
lib/test.rb
```

Utilizando la estructura ACL devuelta por el método `get_acl_access_data` y comprobándola sobre la lista de archivos de cada confirmación de cambios, puedes determinar si el usuario tiene o no permiso para enviar dichos cambios:

```
# only allows certain users to modify certain subdirectories in a project
def check_directory_perms
  access = get_acl_access_data('acl')

  # see if anyone is trying to push something they can't
  new_commits = `git rev-list #{$oldrev}..#{$newrev}`.split("\n")
  new_commits.each do |rev|
    files_modified = `git log -1 --name-only --pretty=format:'' #{rev}`.split("\n")
    files_modified.each do |path|
      next if path.size == 0
      has_file_access = false
      access[$user].each do |access_path|
        if !access_path # user has access to everything
          || (path.start_with? access_path) # access to this path
          has_file_access = true
        end
      end
      if !has_file_access
        puts "[POLICY] You do not have access to push to #{path}"
        exit 1
      end
    end
  end
end

check_directory_perms
```

Se puede obtener una lista de los nuevos commits a enviar con `git rev-list`. Para cada uno de ellos, puedes ver qué ficheros se quieren modificar y

asegurarse que el usuario que está enviando los ficheros tiene acceso a todos ellos.

Desde este momento, los usuarios ya no podrán subir cambios con mensajes de confirmación que no cumplen las reglas, o cuando intenten modificar ficheros a los que no tienen acceso.

COMPROBACIÓN

Si lanzas `chmod u+x .git/hooks/update`, siendo este el fichero en el que hemos introducido el código anterior, y probamos a subir un commit con un mensaje que no cumple las reglas, verás algo como esto:

```
$ git push -f origin master
Counting objects: 5, done.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 323 bytes, done.
Total 3 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
Enforcing Policies...
(refs/heads/master) (8338c5) (c5b616)
[POLICY] Your message is not formatted correctly
error: hooks/update exited with error code 1
error: hook declined to update refs/heads/master
To git@gitserver:project.git
 ! [remote rejected] master -> master (hook declined)
error: failed to push some refs to 'git@gitserver:project.git'
```

Hay un par de cosas interesantes aquí. Lo primero es que ves dónde empieza la ejecución del enganche.

```
Enforcing Policies...
(refs/heads/master) (fb8c72) (c56860)
```

Recuerda que imprimías esto al comienzo del script. Todo lo que el script imprima sobre la salida estándar (`stdout`) se enviará al cliente.

Lo siguiente que ves es el mensaje de error.

```
[POLICY] Your message is not formatted correctly
error: hooks/update exited with error code 1
error: hook declined to update refs/heads/master
```

La primera línea la imprime tu script, las otras dos son las que usa Git para decirte que el script finalizó con error (devuelve un valor distinto de cero) y que está rechazando tu envío. Por último, tienes esto:

```
To git@gitserver:project.git
! [remote rejected] master -> master (hook declined)
error: failed to push some refs to 'git@gitserver:project.git'
```

Verás un mensaje de rechazo remoto para cada referencia que tu script ha rechazado, y te dice qué fue rechazado explícitamente debido al fallo del script de enganche.

Y más aún, si alguien intenta realizar un envío, en el que haya confirmaciones de cambio que afecten a ficheros a los que ese usuario no tiene acceso, verá algo similar. Por ejemplo, si un documentalista intenta tocar algo de la carpeta `lib`, verá esto:

```
[POLICY] You do not have access to push to lib/test.rb
```

Y eso es todo. De ahora en adelante, en tanto en cuando el script `update` esté presente y sea ejecutable, tu repositorio nunca se verá perjudicado, nunca tendrá un mensaje de confirmación de cambios sin tu plantilla y tus usuarios estarán controlados.

Puntos de enganche del lado cliente

Lo malo del sistema descrito en la sección anterior pueden ser los lamentos que inevitablemente se van a producir cuando los envíos de tus usuarios sean rechazados. Ver rechazado en el último minuto su tan cuidadosamente preparado trabajo, puede ser realmente frustrante. Y, aún peor, tener que reescribir su histórico para corregirlo puede ser un auténtico calvario.

La solución a este dilema es el proporcionarles algunos enganches (`hook`) del lado cliente, para que les avisen cuando están trabajando en algo que el servidor va a rechazarles. De esta forma, pueden corregir los problemas antes de confirmar cambios y antes de que se conviertan en algo realmente complicado de arreglar. Debido a que estos enganches no se transfieren junto con el clonado de un proyecto, tendrás que distribuirlos de alguna otra manera. Y luego pedir a tus usuarios que se los copien a sus carpetas `.git/hooks` y los hagan ejecutables. Puedes distribuir esos enganches dentro del mismo proyecto o en un proyecto separado. Pero no hay modo de implementarlos automáticamente.

Para empezar, se necesita chequear el mensaje de confirmación inmediatamente antes de cada confirmación de cambios, para asegurarse de que el servidor no los rechazará debido a un mensaje mal formateado. Para ello, se añade el enganche `commit-msg`. Comparando el mensaje del archivo pasado como primer argumento con el mensaje patrón, puedes obligar a Git a abortar la confirmación de cambios (`commit`) en caso de no coincidir ambos:

```
#!/usr/bin/env ruby
message_file = ARGV[0]
message = File.read(message_file)

$regex = /\[ref: (\d+)\]/

if !$regex.match(message)
  puts "[POLICY] Your message is not formatted correctly"
  exit 1
end
```

Si este script está en su sitio (el archivo `.git/hooks/commit-msg`) y es ejecutable, al confirmar cambios con un mensaje inapropiado, verás algo así como:

```
$ git commit -am 'test'
[POLICY] Your message is not formatted correctly
```

Y la confirmación no se llevará a cabo. Sin embargo, si el mensaje está formateado adecuadamente, Git te permitirá confirmar cambios:

```
$ git commit -am 'test [ref: 132]'
[master e05c914] test [ref: 132]
  1 file changed, 1 insertions(+), 0 deletions(-)
```

A continuación, se necesita también asegurarse de no estar modificando archivos fuera del alcance de tus permisos. Si la carpeta `.git` de tu proyecto contiene una copia del archivo de control de accesos (ACL) utilizada previamente, este script pre-commit podrá comprobar los límites:

```
#!/usr/bin/env ruby

$user      = ENV['USER']

# [ insert acl_access_data method from above ]

# only allows certain users to modify certain subdirectories in a project
def check_directory_perms
  access = get_acl_access_data('.git/acl')

  files_modified = `git diff-index --cached --name-only HEAD`.split("\n")
  files_modified.each do |path|
    next if path.size == 0
    has_file_access = false
    access[$user].each do |access_path|
```

```

if !access_path || (path.index(access_path) == 0)
  has_file_access = true
end
if !has_file_access
  puts "[POLICY] You do not have access to push to #{path}"
  exit 1
end
end
end

check_directory_perms

```

Este es un script prácticamente igual al del lado servidor. Pero con dos importantes diferencias. La primera es que el archivo ACL está en otra ubicación, debido a que el script corre desde tu carpeta de trabajo y no desde la carpeta de Git. Esto obliga a cambiar la ubicación del archivo ACL de aquí:

```
access = get_acl_access_data('acl')
```

a este otro sitio:

```
access = get_acl_access_data('.git/acl')
```

La segunda diferencia es la forma de listar los archivos modificados. Debido a que el método del lado servidor utiliza el registro de confirmaciones de cambio, pero, sin embargo, aquí la confirmación no se ha registrado aún, la lista de archivos se ha de obtener desde el área de preparación (staging area). En lugar de:

```
files_modified = `git log -1 --name-only --pretty=format:'' #{ref}`
```

tenemos que utilizar:

```
files_modified = `git diff-index --cached --name-only HEAD`
```

Estas dos son las únicas diferencias; en todo lo demás, el script funciona de la misma manera. Es necesario advertir de que se espera que trabajes localmente con el mismo usuario con el que enviarás (push) a la máquina remota. Si no fuera así, tendrás que ajustar manualmente la variable \$user.

El último aspecto a comprobar es el de no intentar enviar referencias que no sean de avance-rápido. Pero esto es algo más raro que suceda. Para tener una referencia que no sea de avance-rápido, tienes que haber reorganizado (rebase) una confirmación de cambios (commit) ya enviada anteriormente, o tienes que estar tratando de enviar una rama local distinta sobre la misma rama remota.

De todas formas, el único aspecto accidental que puede ser interesante capturar son los intentos de reorganizar confirmaciones de cambios ya enviadas. El servidor te avisará de que no puedes enviar ningún no-avance-rápido, y el engranaje te impedirá cualquier envío forzado.

Este es un ejemplo de script previo a reorganización que lo puedes comprobar. Con la lista de confirmaciones de cambio que estás a punto de reescribir, las comprueba por si alguna de ellas existe en alguna de tus referencias remotas. Si encuentra alguna, aborta la reorganización:

```
#!/usr/bin/env ruby

base_branch = ARGV[0]
if ARGV[1]
  topic_branch = ARGV[1]
else
  topic_branch = "HEAD"
end

target_shas = `git rev-list #{base_branch}..#{topic_branch}`.split("\n")
remote_refs = `git branch -r`.split("\n").map { |r| r.strip }

target_shas.each do |sha|
  remote_refs.each do |remote_ref|
    shas_pushed = `git rev-list ^#{sha}@ refs/remotes/#{remote_ref}`
    if shas_pushed.split("\n").include?(sha)
      puts "[POLICY] Commit #{sha} has already been pushed to #{remote_ref}"
      exit 1
    end
  end
end
```

Este script utiliza una sintaxis no contemplada en la sección de Selección de Revisiones del capítulo 6. La lista de confirmaciones de cambio previamente enviadas, se comprueba con:

```
`git rev-list ^#{sha}@ refs/remotes/#{remote_ref}`
```

La sintaxis *SHA*[@] recupera todos los padres de esa confirmación de cambios (commit). Estás mirando por cualquier confirmación que se pueda alcanzar desde la última en la parte remota, pero que no se pueda alcanzar desde ninguno de los padres de cualquiera de las claves SHA que estás intentando enviar. Es decir, confirmaciones de avance-rápido.

La mayor pega de este sistema es el que puede llegar a ser muy lento; y muchas veces es innecesario, ya que el propio servidor te va a avisar y te impedirá el envío, siempre y cuando no intentes forzar dicho envío con la opción *-f*. De

todas formas, es un ejercicio interesante. Y, en teoría al menos, pude ayudarte a evitar reorganizaciones que luego tengas de echar para atrás y arreglarlas.

Recapitulación

Se han visto las principales vías por donde puedes personalizar tanto tu cliente como tu servidor Git para que se ajusten a tu forma de trabajar y a tus proyectos. Has aprendido todo tipo de ajustes de configuración, atributos basados en archivos e incluso enganches (hooks). Y has preparado un ejemplo de servidor con mecanismos para asegurar políticas determinadas. A partir de ahora estás listo para encajar Git en prácticamente cualquier flujo de trabajo que puedas imaginar.

Git and Other Systems

The world isn't perfect. Usually, you can't immediately switch every project you come in contact with to Git. Sometimes you're stuck on a project using another VCS, and wish it was Git. We'll spend the first part of this chapter learning about ways to use Git as a client when the project you're working on is hosted in a different system.

At some point, you may want to convert your existing project to Git. The second part of this chapter covers how to migrate your project into Git from several specific systems, as well as a method that will work if no pre-built import tool exists.

Git as a Client

Git provides such a nice experience for developers that many people have figured out how to use it on their workstation, even if the rest of their team is using an entirely different VCS. There are a number of these adapters, called "bridges," available. Here we'll cover the ones you're most likely to run into in the wild.

Git and Subversion

A large fraction of open source development projects and a good number of corporate projects use Subversion to manage their source code. It's been around for more than a decade, and for most of that time was the *de facto* VCS choice for open-source projects. It's also very similar in many ways to CVS, which was the big boy of the source-control world before that.

One of Git's great features is a bidirectional bridge to Subversion called `git svn`. This tool allows you to use Git as a valid client to a Subversion server, so you can use all the local features of Git and then push to a Subversion server as if you were using Subversion locally. This means you can do local branching and merging, use the staging area, use rebasing and cherry-picking, and so on,

while your collaborators continue to work in their dark and ancient ways. It's a good way to sneak Git into the corporate environment and help your fellow developers become more efficient while you lobby to get the infrastructure changed to support Git fully. The Subversion bridge is the gateway drug to the DVCS world.

GIT SVN

The base command in Git for all the Subversion bridging commands is `git svn`. It takes quite a few commands, so we'll show the most common while going through a few simple workflows.

It's important to note that when you're using `git svn`, you're interacting with Subversion, which is a system that works very differently from Git. Although you **can** do local branching and merging, it's generally best to keep your history as linear as possible by rebasing your work, and avoiding doing things like simultaneously interacting with a Git remote repository.

Don't rewrite your history and try to push again, and don't push to a parallel Git repository to collaborate with fellow Git developers at the same time. Subversion can have only a single linear history, and confusing it is very easy. If you're working with a team, and some are using SVN and others are using Git, make sure everyone is using the SVN server to collaborate – doing so will make your life easier.

SETTING UP

To demonstrate this functionality, you need a typical SVN repository that you have write access to. If you want to copy these examples, you'll have to make a writeable copy of my test repository. In order to do that easily, you can use a tool called `svnsync` that comes with Subversion. For these tests, we created a new Subversion repository on Google Code that was a partial copy of the `protobuf` project, which is a tool that encodes structured data for network transmission.

To follow along, you first need to create a new local Subversion repository:

```
$ mkdir /tmp/test-svn  
$ svnadmin create /tmp/test-svn
```

Then, enable all users to change revprops – the easy way is to add a `pre-revprop-change` script that always exits 0:

```
$ cat /tmp/test-svn/hooks/pre-revprop-change
#!/bin/sh
exit 0;
$ chmod +x /tmp/test-svn/hooks/pre-revprop-change
```

You can now sync this project to your local machine by calling `svnsync init` with the to and from repositories.

```
$ svnsync init file:///tmp/test-svn \
http://progit-example.googlecode.com/svn/
```

This sets up the properties to run the sync. You can then clone the code by running

```
$ svnsync sync file:///tmp/test-svn
Committed revision 1.
Copied properties for revision 1.
Transmitting file data .....[...]
Committed revision 2.
Copied properties for revision 2.
[...]
```

Although this operation may take only a few minutes, if you try to copy the original repository to another remote repository instead of a local one, the process will take nearly an hour, even though there are fewer than 100 commits. Subversion has to clone one revision at a time and then push it back into another repository – it's ridiculously inefficient, but it's the only easy way to do this.

GETTING STARTED

Now that you have a Subversion repository to which you have write access, you can go through a typical workflow. You'll start with the `git svn clone` command, which imports an entire Subversion repository into a local Git repository. Remember that if you're importing from a real hosted Subversion repository, you should replace the `file:///tmp/test-svn` here with the URL of your Subversion repository:

```
$ git svn clone file:///tmp/test-svn -T trunk -b branches -t tags
Initialized empty Git repository in /private/tmp/progit/test-svn/.git/
r1 = dcfb5891860124cc2e8cc616cded42624897125 (refs/remotes/origin/trunk)
A m4/acx_pthread.m4
```

```

A  m4/stl_hash.m4
A  java/src/test/java/com/google/protobuf/UnknownFieldSetTest.java
A  java/src/test/java/com/google/protobuf/WireFormatTest.java
...
r75 = 556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae (refs/remotes/origin/trunk)
Found possible branch point: file:///tmp/test-svn/trunk => file:///tmp/test-svn/b...
Found branch parent: (refs/remotes/origin/my-calc-branch) 556a3e1e7ad1fde0a32823fc...
Following parent with do_switch
Successfully followed parent
r76 = 0fb585761df569eaecd8146c71e58d70147460a2 (refs/remotes/origin/my-calc-branch)
Checked out HEAD:
  file:///tmp/test-svn/trunk r75

```

This runs the equivalent of two commands – `git svn init` followed by `git svn fetch` – on the URL you provide. This can take a while. The test project has only about 75 commits and the codebase isn’t that big, but Git has to check out each version, one at a time, and commit it individually. For a project with hundreds or thousands of commits, this can literally take hours or even days to finish.

The `-T trunk -b branches -t tags` part tells Git that this Subversion repository follows the basic branching and tagging conventions. If you name your trunk, branches, or tags differently, you can change these options. Because this is so common, you can replace this entire part with `-s`, which means standard layout and implies all those options. The following command is equivalent:

```
$ git svn clone file:///tmp/test-svn -s
```

At this point, you should have a valid Git repository that has imported your branches and tags:

```

$ git branch -a
* master
  remotes/origin/my-calc-branch
  remotes/origin/tags/2.0.2
  remotes/origin/tags/release-2.0.1
  remotes/origin/tags/release-2.0.2
  remotes/origin/tags/release-2.0.2rc1
  remotes/origin/trunk

```

Note how this tool manages Subversion tags as remote refs. Let’s take a closer look with the Git plumbing command `show-ref`:

```
$ git show-ref
556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae refs/heads/master
0fb585761df569eaecd8146c71e58d70147460a2 refs/remotes/origin/my-calc-branch
bfd2d79303166789fc73af4046651a4b35c12f0b refs/remotes/origin/tags/2.0.2
285c2b2e36e467dd4d91c8e3c0c0e1750b3fe8ca refs/remotes/origin/tags/release-2.0.1
cbda99cb45d9abcb9793db1d4f70ae562a969f1e refs/remotes/origin/tags/release-2.0.2
a9f074aa89e826d6f9d30808ce5ae3ffe711fed4 refs/remotes/origin/tags/release-2.0.2rc1
556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae refs/remotes/origin/trunk
```

Git doesn't do this when it clones from a Git server; here's what a repository with tags looks like after a fresh clone:

```
$ git show-ref
c3dcbe8488c6240392e8a5d7553bbffcb0f94ef0 refs/remotes/origin/master
32ef1d1c7cc8c603ab78416262cc421b80a8c2df refs/remotes/origin/branch-1
75f703a3580a9b81ead89fe1138e6da858c5ba18 refs/remotes/origin/branch-2
23f8588dde934e8f33c263c6d8359b2ae095f863 refs/tags/v0.1.0
7064938bd5e7ef47bfd79a685a62c1e2649e2ce7 refs/tags/v0.2.0
6dc09b5b57875f334f61aeb6d95e2e4193db5e refs/tags/v1.0.0
```

Git fetches the tags directly into `refs/tags`, rather than treating them remote branches.

COMMITTING BACK TO SUBVERSION

Now that you have a working repository, you can do some work on the project and push your commits back upstream, using Git effectively as a SVN client. If you edit one of the files and commit it, you have a commit that exists in Git locally that doesn't exist on the Subversion server:

```
$ git commit -am 'Adding git-svn instructions to the README'
[master 4af61fd] Adding git-svn instructions to the README
 1 file changed, 5 insertions(+)
```

Next, you need to push your change upstream. Notice how this changes the way you work with Subversion – you can do several commits offline and then push them all at once to the Subversion server. To push to a Subversion server, you run the `git svn dcommit` command:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
 M README.txt
```

```
Committed r77
  M  README.txt
r77 = 95e0222ba6399739834380eb10afcd73e0670bc5 (refs/remotes/origin/trunk)
No changes between 4af61fd05045e07598c553167e0f31c84fd6ffe1 and refs/remotes/origin/trunk
Resetting to the latest refs/remotes/origin/trunk
```

This takes all the commits you've made on top of the Subversion server code, does a Subversion commit for each, and then rewrites your local Git commit to include a unique identifier. This is important because it means that all the SHA-1 checksums for your commits change. Partly for this reason, working with Git-based remote versions of your projects concurrently with a Subversion server isn't a good idea. If you look at the last commit, you can see the new `git-svn-id` that was added:

```
$ git log -1
commit 95e0222ba6399739834380eb10afcd73e0670bc5
Author: ben <ben@0b684db3-b064-4277-89d1-21af03df0a68>
Date:   Thu Jul 24 03:08:36 2014 +0000

        Adding git-svn instructions to the README

    git-svn-id: file:///tmp/test-svn/trunk@77 0b684db3-b064-4277-89d1-21af03df0a68
```

Notice that the SHA-1 checksum that originally started with `4af61fd` when you committed now begins with `95e0222`. If you want to push to both a Git server and a Subversion server, you have to push (`dcommit`) to the Subversion server first, because that action changes your commit data.

PULLING IN NEW CHANGES

If you're working with other developers, then at some point one of you will push, and then the other one will try to push a change that conflicts. That change will be rejected until you merge in their work. In `git svn`, it looks like this:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...

ERROR from SVN:
Transaction is out of date: File '/trunk/README.txt' is out of date
W: d5837c4b461b7c0e018b49d12398769d2bfc240a and refs/remotes/origin/trunk differ,
:100644 100644 f414c433af0fd6734428cf9d2a9fd8ba00ada145 c80b6127dd04f5fcda218730dc
Current branch master is up to date.
```

```
ERROR: Not all changes have been committed into SVN, however the committed
ones (if any) seem to be successfully integrated into the working tree.
Please see the above messages for details.
```

To resolve this situation, you can run `git svn rebase`, which pulls down any changes on the server that you don't have yet and rebases any work you have on top of what is on the server:

```
$ git svn rebase
Committing to file:///tmp/test-svn/trunk ...

ERROR from SVN:
Transaction is out of date: File '/trunk/README.txt' is out of date
W: eaa029d99f87c5c822c5c29039d1911ff32ef46 and refs/remotes/origin/trunk differ, using reba
:100644 100644 65536c6e30d263495c17d781962cff12422693a b34372b25ccf4945fe5658fa381b075045e7
First, rewinding head to replay your work on top of it...
Applying: update foo
Using index info to reconstruct a base tree...
M README.txt
Falling back to patching base and 3-way merge...
Auto-merging README.txt
ERROR: Not all changes have been committed into SVN, however the committed
ones (if any) seem to be successfully integrated into the working tree.
Please see the above messages for details.
```

Now, all your work is on top of what is on the Subversion server, so you can successfully `dcommit`:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
M README.txt
Committed r85
M README.txt
r85 = 9c29704cc0bbbed7bd58160cfb66cb9191835cd8 (refs/remotes/origin/trunk)
No changes between 5762f56732a958d6cfda681b661d2a239cc53ef5 and refs/remotes/origin/trunk
Resetting to the latest refs/remotes/origin/trunk
```

Note that unlike Git, which requires you to merge upstream work you don't yet have locally before you can push, `git svn` makes you do that only if the changes conflict (much like how Subversion works). If someone else pushes a change to one file and then you push a change to another file, your `dcommit` will work fine:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
    M  configure.ac
Committed r87
    M  autogen.sh
r86 = d8450bab8a77228a644b7dc0e95977ffc61adff7 (refs/remotes/origin/trunk)
    M  configure.ac
r87 = f3653ea40cb4e26b6281cec102e35dcba1fe17c4 (refs/remotes/origin/trunk)
W: a0253d06732169107aa020390d9fefd2b1d92806 and refs/remotes/origin/trunk differ,
:100755 100755 efa5a59965fbbb5b2b0a12890f1b351bb5493c18 e757b59a9439312d80d5d43bb6
First, rewinding head to replay your work on top of it...
```

This is important to remember, because the outcome is a project state that didn't exist on either of your computers when you pushed. If the changes are incompatible but don't conflict, you may get issues that are difficult to diagnose. This is different than using a Git server – in Git, you can fully test the state on your client system before publishing it, whereas in SVN, you can't ever be certain that the states immediately before commit and after commit are identical.

You should also run this command to pull in changes from the Subversion server, even if you're not ready to commit yourself. You can run `git svn fetch` to grab the new data, but `git svn rebase` does the fetch and then updates your local commits.

```
$ git svn rebase
    M  autogen.sh
r88 = c9c5f83c64bd755368784b444bc7a0216cc1e17b (refs/remotes/origin/trunk)
First, rewinding head to replay your work on top of it...
Fast-forwarded master to refs/remotes/origin/trunk.
```

Running `git svn rebase` every once in a while makes sure your code is always up to date. You need to be sure your working directory is clean when you run this, though. If you have local changes, you must either stash your work or temporarily commit it before running `git svn rebase` – otherwise, the command will stop if it sees that the rebase will result in a merge conflict.

GIT BRANCHING ISSUES

When you've become comfortable with a Git workflow, you'll likely create topic branches, do work on them, and then merge them in. If you're pushing to a Subversion server via `git svn`, you may want to rebase your work onto a single branch each time instead of merging branches together. The reason to pre-

fer rebasing is that Subversion has a linear history and doesn't deal with merges like Git does, so `git svn` follows only the first parent when converting the snapshots into Subversion commits.

Suppose your history looks like the following: you created an `experiment` branch, did two commits, and then merged them back into `master`. When you `dcommit`, you see output like this:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
M  CHANGES.txt
Committed r89
M  CHANGES.txt
r89 = 89d492c884ea7c834353563d5d913c6adf933981 (refs/remotes/origin/trunk)
M  COPYING.txt
M  INSTALL.txt
Committed r90
M  INSTALL.txt
M  COPYING.txt
r90 = cb522197870e61467473391799148f6721bcf9a0 (refs/remotes/origin/trunk)
No changes between 71af502c214ba13123992338569f4669877f55fd and refs/remotes/origin/trunk
Resetting to the latest refs/remotes/origin/trunk
```

Running `dcommit` on a branch with merged history works fine, except that when you look at your Git project history, it hasn't rewritten either of the commits you made on the `experiment` branch – instead, all those changes appear in the SVN version of the single merge commit.

When someone else clones that work, all they see is the merge commit with all the work squashed into it, as though you ran `git merge --squash`; they don't see the commit data about where it came from or when it was committed.

SUBVERSION BRANCHING

Branching in Subversion isn't the same as branching in Git; if you can avoid using it much, that's probably best. However, you can create and commit to branches in Subversion using `git svn`.

CREATING A NEW SVN BRANCH

To create a new branch in Subversion, you run `git svn branch [branch-name]`:

```
$ git svn branch opera
Copying file:///tmp/test-svn/trunk at r90 to file:///tmp/test-svn/branches/opera...
Found possible branch point: file:///tmp/test-svn/trunk => file:///tmp/test-svn/branches/opera
Found branch parent: (refs/remotes/origin/opera) cb522197870e61467473391799148f672
Following parent with do_switch
Successfully followed parent
r91 = f1b64a3855d3c8dd84ee0ef10fa89d27f1584302 (refs/remotes/origin/opera)
```

This does the equivalent of the `svn copy trunk branches/opera` command in Subversion and operates on the Subversion server. It's important to note that it doesn't check you out into that branch; if you commit at this point, that commit will go to `trunk` on the server, not `opera`.

SWITCHING ACTIVE BRANCHES

Git figures out what branch your dcommits go to by looking for the tip of any of your Subversion branches in your history – you should have only one, and it should be the last one with a `git-svn-id` in your current branch history.

If you want to work on more than one branch simultaneously, you can set up local branches to `dcommit` to specific Subversion branches by starting them at the imported Subversion commit for that branch. If you want an `opera` branch that you can work on separately, you can run

```
$ git branch opera remotes/origin/opera
```

Now, if you want to merge your `opera` branch into `trunk` (your `master` branch), you can do so with a normal `git merge`. But you need to provide a descriptive commit message (via `-m`), or the merge will say “Merge branch `opera`” instead of something useful.

Remember that although you're using `git merge` to do this operation, and the merge likely will be much easier than it would be in Subversion (because Git will automatically detect the appropriate merge base for you), this isn't a normal Git merge commit. You have to push this data back to a Subversion server that can't handle a commit that tracks more than one parent; so, after you push it up, it will look like a single commit that squashed in all the work of another branch under a single commit. After you merge one branch into another, you can't easily go back and continue working on that branch, as you normally can in Git. The `dcommit` command that you run erases any information that says what branch was merged in, so subsequent merge-base calculations will be wrong – the `dcommit` makes your `git merge` result look like you ran `git`

`merge --squash`. Unfortunately, there's no good way to avoid this situation – Subversion can't store this information, so you'll always be crippled by its limitations while you're using it as your server. To avoid issues, you should delete the local branch (in this case, `opera`) after you merge it into `trunk`.

SUBVERSION COMMANDS

The `git svn` toolset provides a number of commands to help ease the transition to Git by providing some functionality that's similar to what you had in Subversion. Here are a few commands that give you what Subversion used to.

SVN Style History

If you're used to Subversion and want to see your history in SVN output style, you can run `git svn log` to view your commit history in SVN formatting:

```
$ git svn log
-----
r87 | schacon | 2014-05-02 16:07:37 -0700 (Sat, 02 May 2014) | 2 lines
autogen change

-----
r86 | schacon | 2014-05-02 16:00:21 -0700 (Sat, 02 May 2014) | 2 lines
Merge branch 'experiment'

-----
r85 | schacon | 2014-05-02 16:00:09 -0700 (Sat, 02 May 2014) | 2 lines
updated the changelog
```

You should know two important things about `git svn log`. First, it works offline, unlike the real `svn log` command, which asks the Subversion server for the data. Second, it only shows you commits that have been committed up to the Subversion server. Local Git commits that you haven't dcommitted don't show up; neither do commits that people have made to the Subversion server in the meantime. It's more like the last known state of the commits on the Subversion server.

SVN Annotation

Much as the `git svn log` command simulates the `svn log` command offline, you can get the equivalent of `svn annotate` by running `git svn blame [FILE]`. The output looks like this:

```
$ git svn blame README.txt
 2 temporal Protocol Buffers - Google's data interchange format
 2 temporal Copyright 2008 Google Inc.
 2 temporal http://code.google.com/apis/protocolbuffers/
 2 temporal
22 temporal C++ Installation - Unix
22 temporal =====
 2 temporal
79 schacon Committing in git-svn.
78 schacon
 2 temporal To build and install the C++ Protocol Buffer runtime and the Protoco
 2 temporal Buffer compiler (protoc) execute the following:
 2 temporal
```

Again, it doesn't show commits that you did locally in Git or that have been pushed to Subversion in the meantime.

SVN Server Information

You can also get the same sort of information that `svn info` gives you by running `git svn info`:

```
$ git svn info
Path: .
URL: https://schacon-test.googlecode.com/svn/trunk
Repository Root: https://schacon-test.googlecode.com/svn
Repository UUID: 4c93b258-373f-11de-be05-5f7a86268029
Revision: 87
Node Kind: directory
Schedule: normal
Last Changed Author: schacon
Last Changed Rev: 87
Last Changed Date: 2009-05-02 16:07:37 -0700 (Sat, 02 May 2009)
```

This is like `blame` and `log` in that it runs offline and is up to date only as of the last time you communicated with the Subversion server.

Ignoring What Subversion Ignores

If you clone a Subversion repository that has `svn:ignore` properties set anywhere, you'll likely want to set corresponding `.gitignore` files so you don't accidentally commit files that you shouldn't. `git svn` has two commands to help with this issue. The first is `git svn create-ignore`, which automatically creates corresponding `.gitignore` files for you so your next commit can include them.

The second command is `git svn show-ignore`, which prints to stdout the lines you need to put in a `.gitignore` file so you can redirect the output into your project exclude file:

```
$ git svn show-ignore > .git/info/exclude
```

That way, you don't litter the project with `.gitignore` files. This is a good option if you're the only Git user on a Subversion team, and your teammates don't want `.gitignore` files in the project.

GIT-SVN SUMMARY

The `git svn` tools are useful if you're stuck with a Subversion server, or are otherwise in a development environment that necessitates running a Subversion server. You should consider it crippled Git, however, or you'll hit issues in translation that may confuse you and your collaborators. To stay out of trouble, try to follow these guidelines:

- Keep a linear Git history that doesn't contain merge commits made by `git merge`. Rebase any work you do outside of your mainline branch back onto it; don't merge it in.
- Don't set up and collaborate on a separate Git server. Possibly have one to speed up clones for new developers, but don't push anything to it that doesn't have a `git-svn-id` entry. You may even want to add a `pre-receive` hook that checks each commit message for a `git-svn-id` and rejects pushes that contain commits without it.

If you follow those guidelines, working with a Subversion server can be more bearable. However, if it's possible to move to a real Git server, doing so can gain your team a lot more.

Git and Mercurial

The DVCS universe is larger than just Git. In fact, there are many other systems in this space, each with their own angle on how to do distributed version control correctly. Apart from Git, the most popular is Mercurial, and the two are very similar in many respects.

The good news, if you prefer Git's client-side behavior but are working with a project whose source code is controlled with Mercurial, is that there's a way to use Git as a client for a Mercurial-hosted repository. Since the way Git talks to server repositories is through remotes, it should come as no surprise that this

bridge is implemented as a remote helper. The project’s name is git-remote-hg, and it can be found at <https://github.com/felipec/git-remote-hg>.

GIT-REMOTE-HG

First, you need to install git-remote-hg. This basically entails dropping its file somewhere in your path, like so:

```
$ curl -o ~/bin/git-remote-hg \
  https://raw.githubusercontent.com/felipec/git-remote-hg/master/git-remote-hg
$ chmod +x ~/bin/git-remote-hg
```

...assuming `~/bin` is in your `$PATH`. Git-remote-hg has one other dependency: the `mercurial` library for Python. If you have Python installed, this is as simple as:

```
$ pip install mercurial
```

(If you don’t have Python installed, visit <https://www.python.org/> and get it first.)

The last thing you’ll need is the Mercurial client. Go to <http://mercurial.selenic.com/> and install it if you haven’t already.

Now you’re ready to rock. All you need is a Mercurial repository you can push to. Fortunately, every Mercurial repository can act this way, so we’ll just use the “hello world” repository everyone uses to learn Mercurial:

```
$ hg clone http://selenic.com/repo/hello /tmp/hello
```

GETTING STARTED

Now that we have a suitable “server-side” repository, we can go through a typical workflow. As you’ll see, these two systems are similar enough that there isn’t much friction.

As always with Git, first we clone:

```
$ git clone hg:///tmp/hello /tmp/hello-git
$ cd /tmp/hello-git
$ git log --oneline --graph --decorate
```

```
* ac7955c (HEAD, origin/master, origin/branches/default, origin/HEAD, refs/hg/origin/branches/default)
* 65bb417 Create a standard "hello, world" program
```

You'll notice that working with a Mercurial repository uses the standard `git clone` command. That's because `git-remote-hg` is working at a fairly low level, using a similar mechanism to how Git's HTTP/S protocol is implemented (remote helpers). Since Git and Mercurial are both designed for every client to have a full copy of the repository history, this command makes a full clone, including all the project's history, and does it fairly quickly.

The log command shows two commits, the latest of which is pointed to by a whole slew of refs. It turns out some of these aren't actually there. Let's take a look at what's actually in the `.git` directory:

```
$ tree .git/refs
.git/refs
├── heads
│   └── master
├── hg
│   └── origin
│       ├── bookmarks
│       │   └── master
│       └── branches
│           └── default
└── notes
    └── hg
└── remotes
    └── origin
        └── HEAD
└── tags
```

9 directories, 5 files

`Git-remote-hg` is trying to make things more idiomatically Git-esque, but under the hood it's managing the conceptual mapping between two slightly different systems. The `refs/hg` directory is where the actual remote refs are stored. For example, the `refs/hg/origin/branches/default` is a Git ref file that contains the SHA-1 starting with “`ac7955c`”, which is the commit that `master` points to. So the `refs/hg` directory is kind of like a fake `refs/remotes/origin`, but it has the added distinction between bookmarks and branches.

The `notes/hg` file is the starting point for how `git-remote-hg` maps Git commit hashes to Mercurial changeset IDs. Let's explore a bit:

```
$ cat notes/hg
d4c10386...

$ git cat-file -p d4c10386...
tree 1781c96...
author remote-hg <> 1408066400 -0800
committer remote-hg <> 1408066400 -0800

Notes for master

$ git ls-tree 1781c96...
100644 blob ac9117f... 65bb417...
100644 blob 485e178... ac7955c...

$ git cat-file -p ac9117f
0a04b987be5ae354b710cefeba0e2d9de7ad41a9
```

So `refs/notes/hg` points to a tree, which in the Git object database is a list of other objects with names. `git ls-tree` outputs the mode, type, object hash, and filename for items inside a tree. Once we dig down to one of the tree items, we find that inside it is a blob named “`ac9117f`” (the SHA-1 hash of the commit pointed to by `master`), with contents “`0a04b98`” (which is the ID of the Mercurial changeset at the tip of the `default` branch).

The good news is that we mostly don’t have to worry about all of this. The typical workflow won’t be very different from working with a Git remote.

There’s one more thing we should attend to before we continue: ignores. Mercurial and Git use a very similar mechanism for this, but it’s likely you don’t want to actually commit a `.gitignore` file into a Mercurial repository. Fortunately, Git has a way to ignore files that’s local to an on-disk repository, and the Mercurial format is compatible with Git, so you just have to copy it over:

```
$ cp .hgignore .git/info/exclude
```

The `.git/info/exclude` file acts just like a `.gitignore`, but isn’t included in commits.

WORKFLOW

Let’s assume we’ve done some work and made some commits on the `master` branch, and you’re ready to push it to the remote repository. Here’s what our repository looks like right now:

```
$ git log --oneline --graph --decorate
```

- * ba04a2a (HEAD, master) Update makefile
- * d25d16f Goodbye
- * ac7955c (origin/master, origin/branches/default, origin/HEAD, refs/hg/origin/branches/default)
- * 65bb417 Create a standard "hello, world" program

Our `master` branch is two commits ahead of `origin/master`, but those two commits exist only on our local machine. Let's see if anyone else has been doing important work at the same time:

```
$ git fetch
```

```
From hg::/tmp/hello
```

- ac7955c..df85e87 master -> origin/master

- ac7955c..df85e87 branches/default -> origin/branches/default

```
$ git log --oneline --graph --decorate --all
```

- * 7b07969 (refs/notes/hg) Notes for default

- * d4c1038 Notes for master

- * df85e87 (origin/master, origin/branches/default, origin/HEAD, refs/hg/origin/branches/default)

- | * ba04a2a (HEAD, master) Update makefile

- | * d25d16f Goodbye

- | /

- * ac7955c Create a makefile

- * 65bb417 Create a standard "hello, world" program

Since we used the `--all` flag, we see the “notes” refs that are used internally by `git-remote-hg`, but we can ignore them. The rest is what we expected; `origin/master` has advanced by one commit, and our history has now diverged. Unlike the other systems we work with in this chapter, Mercurial is capable of handling merges, so we’re not going to do anything fancy.

```
$ git merge origin/master
```

```
Auto-merging hello.c
```

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

```
hello.c | 2 +-
```

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

```
$ git log --oneline --graph --decorate
```

- * 0c64627 (HEAD, master) Merge remote-tracking branch 'origin/master'

- | \

- | * df85e87 (origin/master, origin/branches/default, origin/HEAD, refs/hg/origin/branches/default)

- | * ba04a2a Update makefile

- | * d25d16f Goodbye

- | /

- * ac7955c Create a makefile

- * 65bb417 Create a standard "hello, world" program

Perfect. We run the tests and everything passes, so we’re ready to share our work with the rest of the team:

```
$ git push
To hg::/tmp/hello
 df85e87..0c64627 master -> master
```

That’s it! If you take a look at the Mercurial repository, you’ll see that this did what we’d expect:

```
$ hg log -G --style compact
o 5[tip]:4,2 dc8fa4f932b8 2014-08-14 19:33 -0700 ben
|\ Merge remote-tracking branch 'origin/master'
|
| o 4 64f27bcef35 2014-08-14 19:27 -0700 ben
| | Update makefile
|
| o 3:1 4256fc29598f 2014-08-14 19:27 -0700 ben
| | Goodbye
|
@ | 2 7db0b4848b3c 2014-08-14 19:30 -0700 ben
| / Add some documentation
|
o 1 82e55d328c8c 2005-08-26 01:21 -0700 mpm
| Create a makefile
|
o 0 0a04b987be5a 2005-08-26 01:20 -0700 mpm
| Create a standard "hello, world" program
```

The changeset numbered 2 was made by Mercurial, and the changesets numbered 3 and 4 were made by git-remote-hg, by pushing commits made with Git.

BRANCHES AND BOOKMARKS

Git has only one kind of branch: a reference that moves when commits are made. In Mercurial, this kind of a reference is called a “bookmark,” and it behaves in much the same way as a Git branch.

Mercurial’s concept of a “branch” is more heavyweight. The branch that a changeset is made on is recorded *with the changeset*, which means it will always be in the repository history. Here’s an example of a commit that was made on the `develop` branch:

```
$ hg log -l 1
changeset:   6:8f65e5e02793
branch:      develop
tag:         tip
user:        Ben Straub <ben@straub.cc>
date:        Thu Aug 14 20:06:38 2014 -0700
summary:     More documentation
```

Note the line that begins with “branch”. Git can’t really replicate this (and doesn’t need to; both types of branch can be represented as a Git ref), but git-remote-hg needs to understand the difference, because Mercurial cares.

Creating Mercurial bookmarks is as easy as creating Git branches. On the Git side:

```
$ git checkout -b featureA
Switched to a new branch 'featureA'
$ git push origin featureA
To hg::/tmp/hello
 * [new branch]      featureA -> featureA
```

That’s all there is to it. On the Mercurial side, it looks like this:

```
$ hg bookmarks
featureA           5:bd5ac26f11f9
$ hg log --style compact -G
@ 6[tip] 8f65e5e02793 2014-08-14 20:06 -0700    ben
| More documentation
|
o 5[featureA]:4,2 bd5ac26f11f9 2014-08-14 20:02 -0700    ben
|\ Merge remote-tracking branch 'origin/master'
|
| o 4 0434aaa6b91f 2014-08-14 20:01 -0700    ben
| | update makefile
|
| o 3:1 318914536c86 2014-08-14 20:00 -0700    ben
| | goodbye
|
o | 2 f098c7f45c4f 2014-08-14 20:01 -0700    ben
|/ Add some documentation
|
o | 1 82e55d328c8c 2005-08-26 01:21 -0700    mpm
| Create a makefile
|
```

```

o 0 0a04b987be5a 2005-08-26 01:20 -0700 mpm
Create a standard "hello, world" program

```

Note the new [featureA] tag on revision 5. These act exactly like Git branches on the Git side, with one exception: you can't delete a bookmark from the Git side (this is a limitation of remote helpers).

You can work on a “heavyweight” Mercurial branch also: just put a branch in the branches namespace:

```

$ git checkout -b branches/permanent
Switched to a new branch 'branches/permanent'
$ vi Makefile
$ git commit -am 'A permanent change'
$ git push origin branches/permanent
To hg::/tmp/hello
 * [new branch] branches/permanent -> branches/permanent

```

Here's what that looks like on the Mercurial side:

```

$ hg branches
permanent                               7:a4529d07aad4
develop                                6:8f65e5e02793
default                                 5:bd5ac26f11f9 (inactive)
$ hg log -G
o changeset: 7:a4529d07aad4
| branch: permanent
| tag: tip
| parent: 5:bd5ac26f11f9
| user: Ben Straub <ben@straub.cc>
| date: Thu Aug 14 20:21:09 2014 -0700
| summary: A permanent change
|
| @ changeset: 6:8f65e5e02793
| / branch: develop
| | user: Ben Straub <ben@straub.cc>
| | date: Thu Aug 14 20:06:38 2014 -0700
| | summary: More documentation
|
o changeset: 5:bd5ac26f11f9
|\ bookmark: featureA
| | parent: 4:0434aaa6b91f
| | parent: 2:f098c7f45c4f
| | user: Ben Straub <ben@straub.cc>
| | date: Thu Aug 14 20:02:21 2014 -0700

```

```
| | summary: Merge remote-tracking branch 'origin/master'  
[...]
```

The branch name “permanent” was recorded with the changeset marked 7. From the Git side, working with either of these branch styles is the same: just checkout, commit, fetch, merge, pull, and push as you normally would. One thing you should know is that Mercurial doesn’t support rewriting history, only adding to it. Here’s what our Mercurial repository looks like after an interactive rebase and a force-push:

```
$ hg log --style compact -G  
o 10[tip] 99611176cbc9 2014-08-14 20:21 -0700 ben  
| A permanent change  
|  
o 9 f23e12f939c3 2014-08-14 20:01 -0700 ben  
| Add some documentation  
|  
o 8:1 c16971d33922 2014-08-14 20:00 -0700 ben  
| goodbye  
|  
| o 7:5 a4529d07aad4 2014-08-14 20:21 -0700 ben  
| | A permanent change  
| |  
| | @ 6 8f65e5e02793 2014-08-14 20:06 -0700 ben  
| | | More documentation  
| | |  
| | o 5[featureA]:4,2 bd5ac26f11f9 2014-08-14 20:02 -0700 ben  
| | | | Merge remote-tracking branch 'origin/master'  
| | | |  
| | | o 4 0434aaa6b91f 2014-08-14 20:01 -0700 ben  
| | | | update makefile  
| | | |  
+--o 3:1 318914536c86 2014-08-14 20:00 -0700 ben  
| | | goodbye  
| | |  
| o 2 f098c7f45c4f 2014-08-14 20:01 -0700 ben  
| | | Add some documentation  
| | |  
o 1 82e55d328c8c 2005-08-26 01:21 -0700 mpm  
| | Create a makefile  
| |  
o 0 0a04b987be5a 2005-08-26 01:20 -0700 mpm  
| | Create a standard "hello, world" program
```

Changesets 8, 9, and 10 have been created and belong to the permanent branch, but the old changesets are still there. This can be **very** confusing for your teammates who are using Mercurial, so try to avoid it.

MERCURIAL SUMMARY

Git and Mercurial are similar enough that working across the boundary is fairly painless. If you avoid changing history that's left your machine (as is generally recommended), you may not even be aware that the other end is Mercurial.

Git and Perforce

Perforce is a very popular version-control system in corporate environments. It's been around since 1995, which makes it the oldest system covered in this chapter. As such, it's designed with the constraints of its day; it assumes you're always connected to a single central server, and only one version is kept on the local disk. To be sure, its features and constraints are well-suited to several specific problems, but there are lots of projects using Perforce where Git would actually work better.

There are two options if you'd like to mix your use of Perforce and Git. The first one we'll cover is the "Git Fusion" bridge from the makers of Perforce, which lets you expose subtrees of your Perforce depot as read-write Git repositories. The second is git-p4, a client-side bridge that lets you use Git as a Perforce client, without requiring any reconfiguration of the Perforce server.

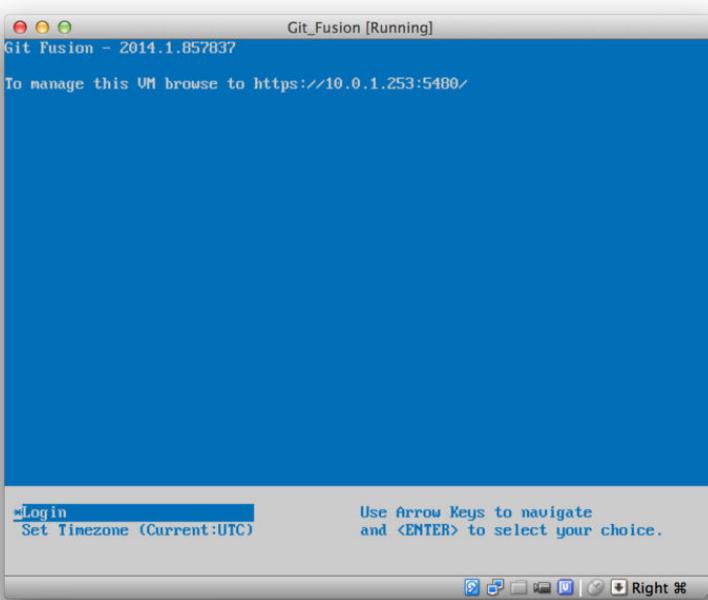
GIT FUSION

Perforce provides a product called Git Fusion (available at <http://www.perforce.com/git-fusion>), which synchronizes a Perforce server with Git repositories on the server side.

Setting Up

For our examples, we'll be using the easiest installation method for Git Fusion, which is downloading a virtual machine that runs the Perforce daemon and Git Fusion. You can get the virtual machine image from <http://www.perforce.com/downloads/Perforce/20-User>, and once it's finished downloading, import it into your favorite virtualization software (we'll use VirtualBox).

Upon first starting the machine, it asks you to customize the password for three Linux users (`root`, `perforce`, and `git`), and provide an instance name, which can be used to distinguish this installation from others on the same network. When that has all completed, you'll see this:

**FIGURE 9-1**

The Git Fusion virtual machine boot screen.

You should take note of the IP address that's shown here, we'll be using it later on. Next, we'll create a Perforce user. Select the "Login" option at the bottom and press enter (or SSH to the machine), and log in as `root`. Then use these commands to create a user:

```
$ p4 -p localhost:1666 -u super user -f john  
$ p4 -p localhost:1666 -u john passwd  
$ exit
```

The first one will open a VI editor to customize the user, but you can accept the defaults by typing :wq and hitting enter. The second one will prompt you to enter a password twice. That's all we need to do with a shell prompt, so exit out of the session.

The next thing you'll need to do to follow along is to tell Git not to verify SSL certificates. The Git Fusion image comes with a certificate, but it's for a domain that won't match your virtual machine's IP address, so Git will reject the HTTPS connection. If this is going to be a permanent installation, consult the Perforce

Git Fusion manual to install a different certificate; for our example purposes, this will suffice:

```
$ export GIT_SSL_NO_VERIFY=true
```

Now we can test that everything is working.

```
$ git clone https://10.0.1.254/Talkhouse
Cloning into 'Talkhouse'...
Username for 'https://10.0.1.254': john
Password for 'https://john@10.0.1.254':
remote: Counting objects: 630, done.
remote: Compressing objects: 100% (581/581), done.
remote: Total 630 (delta 172), reused 0 (delta 0)
Receiving objects: 100% (630/630), 1.22 MiB | 0 bytes/s, done.
Resolving deltas: 100% (172/172), done.
Checking connectivity... done.
```

The virtual-machine image comes equipped with a sample project that you can clone. Here we're cloning over HTTPS, with the `john` user that we created above; Git asks for credentials for this connection, but the credential cache will allow us to skip this step for any subsequent requests.

Fusion Configuration

Once you've got Git Fusion installed, you'll want to tweak the configuration. This is actually fairly easy to do using your favorite Perforce client; just map the `//.git-fusion` directory on the Perforce server into your workspace. The file structure looks like this:

```
$ tree
.
├── objects
│   ├── repos
│   │   └── [...]
│   └── trees
│       └── [...]
|
└── p4gf_config
├── repos
│   └── Talkhouse
│       └── p4gf_config
└── users
    └── p4gf_usermap
```

```
498 directories, 287 files
```

The `objects` directory is used internally by Git Fusion to map Perforce objects to Git and vice versa, you won't have to mess with anything in there. There's a global `p4gf_config` file in this directory, as well as one for each repository – these are the configuration files that determine how Git Fusion behaves. Let's take a look at the file in the root:

```
[repo-creation]
charset = utf8

[git-to-perforce]
change-owner = author
enable-git-branch-creation = yes
enable-swarm-reviews = yes
enable-git-merge-commits = yes
enable-git-submodules = yes
preflight-commit = none
ignore-author-permissions = no
read-permission-check = none
git-merge-avoidance-after-change-num = 12107

[perfcore-to-git]
http-url = none
ssh-url = none

[@features]
imports = False
chunked-push = False
matrix2 = False
parallel-push = False

[authentication]
email-case-sensitivity = no
```

We won't go into the meanings of these flags here, but note that this is just an INI-formatted text file, much like Git uses for configuration. This file specifies the global options, which can then be overridden by repository-specific configuration files, like `repos/Talkhouse/p4gf_config`. If you open this file, you'll see a `[@repo]` section with some settings that are different from the global defaults. You'll also see sections that look like this:

```
[Talkhouse-master]
git-branch-name = master
view = //depot/Talkhouse/main-dev/... ...
```

This is a mapping between a Perforce branch and a Git branch. The section can be named whatever you like, so long as the name is unique. `git-branch-name` lets you convert a depot path that would be cumbersome under Git to a more friendly name. The `view` setting controls how Perforce files are mapped into the Git repository, using the standard view mapping syntax. More than one mapping can be specified, like in this example:

```
[multi-project-mapping]
git-branch-name = master
view = //depot/project1/main/... project1/...
      //depot/project2/mainline/... project2/...
```

This way, if your normal workspace mapping includes changes in the structure of the directories, you can replicate that with a Git repository.

The last file we'll discuss is `users/p4gf_usermap`, which maps Perforce users to Git users, and which you may not even need. When converting from a Perforce changeset to a Git commit, Git Fusion's default behavior is to look up the Perforce user, and use the email address and full name stored there for the author/committer field in Git. When converting the other way, the default is to look up the Perforce user with the email address stored in the Git commit's author field, and submit the changeset as that user (with permissions applying). In most cases, this behavior will do just fine, but consider the following mapping file:

```
john john@example.com "John Doe"
john johnny@appleseed.net "John Doe"
bob employeeX@example.com "Anon X. Mouse"
joe employeeY@example.com "Anon Y. Mouse"
```

Each line is of the format `<user> <email> "<full name>"`, and creates a single user mapping. The first two lines map two distinct email addresses to the same Perforce user account. This is useful if you've created Git commits under several different email addresses (or change email addresses), but want them to be mapped to the same Perforce user. When creating a Git commit from a Perforce changeset, the first line matching the Perforce user is used for Git authorship information.

The last two lines mask Bob and Joe's actual names and email addresses from the Git commits that are created. This is nice if you want to open-source an internal project, but don't want to publish your employee directory to the entire world. Note that the email addresses and full names should be unique, unless you want all the Git commits to be attributed to a single fictional author.

Workflow

Perforce Git Fusion is a two-way bridge between Perforce and Git version control. Let's have a look at how it feels to work from the Git side. We'll assume we've mapped in the "Jam" project using a configuration file as shown above, which we can clone like this:

```
$ git clone https://10.0.1.254/Jam
Cloning into 'Jam'...
Username for 'https://10.0.1.254': john
Password for 'https://ben@10.0.1.254':
remote: Counting objects: 2070, done.
remote: Compressing objects: 100% (1704/1704), done.
Receiving objects: 100% (2070/2070), 1.21 MiB | 0 bytes/s, done.
remote: Total 2070 (delta 1242), reused 0 (delta 0)
Resolving deltas: 100% (1242/1242), done.
Checking connectivity... done.
$ git branch -a
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/master
  remotes/origin/rel2.1
$ git log --oneline --decorate --graph --all
* 0a38c33 (origin/rel2.1) Create Jam 2.1 release branch.
| * d254865 (HEAD, origin/master, origin/HEAD, master) Upgrade to latest metrowerks on Beos
| * bd2f54a Put in fix for jam's NT handle leak.
| * c0f29e7 Fix URL in a jam doc
| * cc644ac Radstone's lynx port.
[...]
```

The first time you do this, it may take some time. What's happening is that Git Fusion is converting all the applicable changesets in the Perforce history into Git commits. This happens locally on the server, so it's relatively fast, but if you have a lot of history, it can still take some time. Subsequent fetches do incremental conversion, so it'll feel more like Git's native speed.

As you can see, our repository looks exactly like any other Git repository you might work with. There are three branches, and Git has helpfully created a local `master` branch that tracks `origin/master`. Let's do a bit of work, and create a couple of new commits:

```
# ...
$ git log --oneline --decorate --graph --all
* cfd46ab (HEAD, master) Add documentation for new feature
* a730d77 Whitespace
* d254865 (origin/master, origin/HEAD) Upgrade to latest metrowerks on Beos -- the Intel one
```

```
* bd2f54a Put in fix for jam's NT handle leak.
[...]
```

We have two new commits. Now let's check if anyone else has been working:

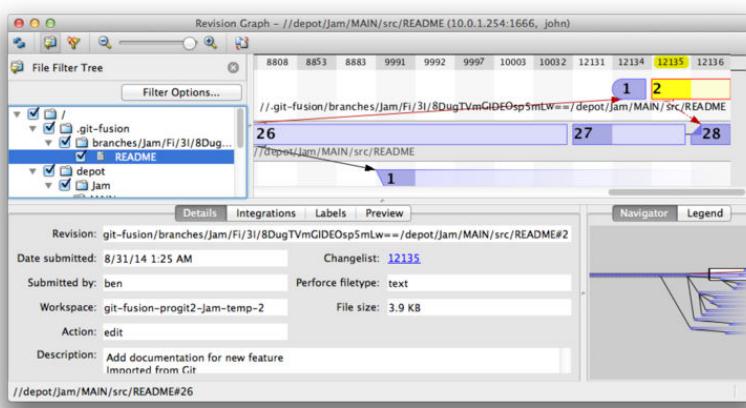
```
$ git fetch
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 2), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From https://10.0.1.254/Jam
    d254865..6afeb15 master      -> origin/master
$ git log --oneline --decorate --graph --all
* 6afeb15 (origin/master, origin/HEAD) Update copyright
| * cfd46ab (HEAD, master) Add documentation for new feature
| * a730d77 Whitespace
|/
* d254865 Upgrade to latest metrowerks on Beos -- the Intel one.
* bd2f54a Put in fix for jam's NT handle leak.
[...]
```

It looks like someone was! You wouldn't know it from this view, but the 6afeb15 commit was actually created using a Perforce client. It just looks like another commit from Git's point of view, which is exactly the point. Let's see how the Perforce server deals with a merge commit:

```
$ git merge origin/master
Auto-merging README
Merge made by the 'recursive' strategy.
 README | 2 ++
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git push
Counting objects: 9, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (9/9), done.
Writing objects: 100% (9/9), 917 bytes | 0 bytes/s, done.
Total 9 (delta 6), reused 0 (delta 0)
remote: Perforce: 100% (3/3) Loading commit tree into memory...
remote: Perforce: 100% (5/5) Finding child commits...
remote: Perforce: Running git fast-export...
remote: Perforce: 100% (3/3) Checking commits...
remote: Perforce: Processing will continue even if connection is closed.
remote: Perforce: 100% (3/3) Copying changelists...
remote: Perforce: Submitting new Git commit objects to Perforce: 4
```

```
To https://10.0.1.254/Jam
6afeb15..89cba2b master -> master
```

Git thinks it worked. Let's take a look at the history of the README file from Perforce's point of view, using the revision graph feature of p4v:

**FIGURE 9-2**

Perforce revision graph resulting from Git push.

If you've never seen this view before, it may seem confusing, but it shows the same concepts as a graphical viewer for Git history. We're looking at the history of the README file, so the directory tree at top left only shows that file as it surfaces in various branches. At top right, we have a visual graph of how different revisions of the file are related, and the big-picture view of this graph is at bottom right. The rest of the view is given to the details view for the selected revision (2 in this case).

One thing to notice is that the graph looks exactly like the one in Git's history. Perforce didn't have a named branch to store the 1 and 2 commits, so it made an "anonymous" branch in the .git-fusion directory to hold it. This will also happen for named Git branches that don't correspond to a named Perforce branch (and you can later map them to a Perforce branch using the configuration file).

Most of this happens behind the scenes, but the end result is that one person on a team can be using Git, another can be using Perforce, and neither of them will know about the other's choice.

Git-Fusion Summary

If you have (or can get) access to your Perforce server, Git Fusion is a great way to make Git and Perforce talk to each other. There's a bit of configuration involved, but the learning curve isn't very steep. This is one of the few sections in this chapter where cautions about using Git's full power will not appear. That's not to say that Perforce will be happy with everything you throw at it – if you try to rewrite history that's already been pushed, Git Fusion will reject it – but Git Fusion tries very hard to feel native. You can even use Git submodules (though they'll look strange to Perforce users), and merge branches (this will be recorded as an integration on the Perforce side).

If you can't convince the administrator of your server to set up Git Fusion, there is still a way to use these tools together.

GIT-P4

Git-p4 is a two-way bridge between Git and Perforce. It runs entirely inside your Git repository, so you won't need any kind of access to the Perforce server (other than user credentials, of course). Git-p4 isn't as flexible or complete a solution as Git Fusion, but it does allow you to do most of what you'd want to do without being invasive to the server environment.

You'll need the p4 tool somewhere in your PATH to work with git-p4. As of this writing, it is freely available at <http://www.perforce.com/downloads/Perforce/zo-User>.

Setting Up

For example purposes, we'll be running the Perforce server from the Git Fusion OVA as shown above, but we'll bypass the Git Fusion server and go directly to the Perforce version control.

In order to use the p4 command-line client (which git-p4 depends on), you'll need to set a couple of environment variables:

```
$ export P4PORT=10.0.1.254:1666
$ export P4USER=john
```

Getting Started

As with anything in Git, the first command is to clone:

```
$ git p4 clone //depot/www/live www-shallow
Importing from //depot/www/live into www-shallow
```

```
Initialized empty Git repository in /private/tmp/www-shallow/.git/
Doing initial import of //depot/www/live/ from revision #head into refs/remotes/p4/master
```

This creates what in Git terms is a “shallow” clone; only the very latest Perforce revision is imported into Git; remember, Perforce isn’t designed to give every revision to every user. This is enough to use Git as a Perforce client, but for other purposes it’s not enough.

Once it’s finished, we have a fully-functional Git repository:

```
$ cd myproject
$ git log --oneline --all --graph --decorate
* 70eaf78 (HEAD, p4/master, p4/HEAD, master) Initial import of //depot/www/live/ from the st
```

Note how there’s a “p4” remote for the Perforce server, but everything else looks like a standard clone. Actually, that’s a bit misleading; there isn’t actually a remote there.

```
$ git remote -v
```

No remotes exist in this repository at all. Git-p4 has created some refs to represent the state of the server, and they look like remote refs to `git log`, but they’re not managed by Git itself, and you can’t push to them.

Workflow

Okay, let’s do some work. Let’s assume you’ve made some progress on a very important feature, and you’re ready to show it to the rest of your team.

```
$ git log --oneline --all --graph --decorate
* 018467c (HEAD, master) Change page title
* c0fb617 Update link
* 70eaf78 (p4/master, p4/HEAD) Initial import of //depot/www/live/ from the state at revision
```

We’ve made two new commits that we’re ready to submit to the Perforce server. Let’s check if anyone else was working today:

```
$ git p4 sync
git p4 sync
Performing incremental import into refs/remotes/p4/master git branch
Depot paths: //depot/www/live/
Import destination: refs/remotes/p4/master
Importing revision 12142 (100%)
$ git log --oneline --all --graph --decorate
```

```
* 75cd059 (p4/master, p4/HEAD) Update copyright
| * 018467c (HEAD, master) Change page title
| * c0fb617 Update link
|/
* 70eaf78 Initial import of //depot/www/live/ from the state at revision #head
```

Looks like they were, and `master` and `p4/master` have diverged. Perforce's branching system is *nothing* like Git's, so submitting merge commits doesn't make any sense. Git-p4 recommends that you rebase your commits, and even comes with a shortcut to do so:

```
$ git p4 rebase
Performing incremental import into refs/remotes/p4/master git branch
Depot paths: //depot/www/live/
No changes to import!
Rebasing the current branch onto remotes/p4/master
First, rewinding head to replay your work on top of it...
Applying: Update link
Applying: Change page title
index.html | 2 ++
1 file changed, 1 insertion(+), 1 deletion(-)
```

You can probably tell from the output, but `git p4 rebase` is a shortcut for `git p4 sync` followed by `git rebase p4/master`. It's a bit smarter than that, especially when working with multiple branches, but this is a good approximation.

Now our history is linear again, and we're ready to contribute our changes back to Perforce. The `git p4 submit` command will try to create a new Perforce revision for every Git commit between `p4/master` and `master`. Running it drops us into our favorite editor, and the contents of the file look something like this:

```
# A Perforce Change Specification.
#
# Change:      The change number. 'new' on a new changelist.
# Date:        The date this specification was last modified.
# Client:      The client on which the changelist was created. Read-only.
# User:        The user who created the changelist.
# Status:      Either 'pending' or 'submitted'. Read-only.
# Type:        Either 'public' or 'restricted'. Default is 'public'.
# Description: Comments about the changelist. Required.
# Jobs:        What opened jobs are to be closed by this changelist.
#              You may delete jobs from this list. (New changelists only.)
# Files:       What opened files from the default changelist are to be added
```

```

#          to this changelist. You may delete files from this list.
#          (New changelists only.)

Change: new

Client: john_bens-mbp_8487

User: john

Status: new

Description:
    Update link

Files:
    //depot/www/live/index.html  # edit

#####
git author ben@straub.cc does not match your p4 account.
Use option --preserve-user to modify authorship.
Variable git-p4.skipUserNameCheck hides this message.
everything below this line is just the diff #####
--- //depot/www/live/index.html 2014-08-31 18:26:05.000000000 0000
+++ /Users/ben/john_bens-mbp_8487/john_bens-mbp_8487/depot/www/live/index.html 2014-08-31
@@ -60,7 +60,7 @@
</td>
<td valign=top>
Source and documentation for
-<a href="http://www.perforce.com/jam/jam.html">
+<a href="jam.html">
Jam/MR</a>,
a software build tool.
</td>
```

This is mostly the same content you'd see by running `p4 submit`, except the stuff at the end which `git-p4` has helpfully included. `Git-p4` tries to honor your Git and Perforce settings individually when it has to provide a name for a commit or changeset, but in some cases you want to override it. For example, if the Git commit you're importing was written by a contributor who doesn't have a Perforce user account, you may still want the resulting changeset to look like they wrote it (and not you).

`Git-p4` has helpfully imported the message from the Git commit as the content for this Perforce changeset, so all we have to do is save and quit, twice (once for each commit). The resulting shell output will look something like this:

```
$ git p4 submit
Perforce checkout for depot path //depot/www/live/ located at /Users/ben/john_bens
Synchronizing p4 checkout...
... - file(s) up-to-date.
Applying dbac45b Update link
//depot/www/live/index.html#4 - opened for edit
Change 12143 created with 1 open file(s).
Submitting change 12143.
Locking 1 files ...
edit //depot/www/live/index.html#5
Change 12143 submitted.
Applying 905ec6a Change page title
//depot/www/live/index.html#5 - opened for edit
Change 12144 created with 1 open file(s).
Submitting change 12144.
Locking 1 files ...
edit //depot/www/live/index.html#6
Change 12144 submitted.
All commits applied!
Performing incremental import into refs/remotes/p4/master git branch
Depot paths: //depot/www/live/
Import destination: refs/remotes/p4/master
Importing revision 12144 (100%)
Rebasing the current branch onto refs/p4/master
First, rewinding head to replay your work on top of it...
$ git log --oneline --all --graph --decorate
* 775a46f (HEAD, p4/master, p4/HEAD, master) Change page title
* 05f1ade Update link
* 75cd059 Update copyright
* 70eaf78 Initial import of //depot/www/live/ from the state at revision #head
```

The result is as though we just did a `git push`, which is the closest analogy to what actually did happen.

Note that during this process every Git commit is turned into a Perforce changeset; if you want to squash them down into a single changeset, you can do that with an interactive rebase before running `git p4 submit`. Also note that the SHA-1 hashes of all the commits that were submitted as changesets have changed; this is because git-p4 adds a line to the end of each commit it converts:

```
$ git log -1
commit 775a46f630d8b46535fc9983cf3ebe6b9aa53145
Author: John Doe <john@example.com>
Date:   Sun Aug 31 10:31:44 2014 -0800

        Change page title
```

```
[git-p4: depot-paths = "//depot/www/live/": change = 12144]
```

What happens if you try to submit a merge commit? Let's give it a try. Here's the situation we've gotten ourselves into:

```
$ git log --oneline --all --graph --decorate
* 3be6fd8 (HEAD, master) Correct email address
*   1dcbf21 Merge remote-tracking branch 'p4/master'
|\ \
| * c4689fc (p4/master, p4/HEAD) Grammar fix
* | cbacd0a Table borders: yes please
* | b4959b6 Trademark
|/
* 775a46f Change page title
* 05f1ade Update link
* 75cd059 Update copyright
* 70eaf78 Initial import of //depot/www/live/ from the state at revision #head
```

The Git and Perforce history diverge after 775a46f. The Git side has two commits, then a merge commit with the Perforce head, then another commit. We're going to try to submit these on top of a single changeset on the Perforce side. Let's see what would happen if we tried to submit now:

```
$ git p4 submit -n
Perforce checkout for depot path //depot/www/live/ located at /Users/ben/john_bens-mbp_8487/
Would synchronize p4 checkout in /Users/ben/john_bens-mbp_8487/john_bens-mbp_8487/depot/www/
Would apply
    b4959b6 Trademark
    cbacd0a Table borders: yes please
    3be6fd8 Correct email address
```

The `-n` flag is short for `--dry-run`, which tries to report what would happen if the `submit` command were run for real. In this case, it looks like we'd be creating three Perforce changesets, which correspond to the three non-merge commits that don't yet exist on the Perforce server. That sounds like exactly what we want, let's see how it turns out:

```
$ git p4 submit
[...]
$ git log --oneline --all --graph --decorate
* dadbd89 (HEAD, p4/master, p4/HEAD, master) Correct email address
* 1b79a80 Table borders: yes please
```

```
* 0097235 Trademark
* c4689fc Grammar fix
* 775a46f Change page title
* 05f1ade Update link
* 75cd059 Update copyright
* 70eaf78 Initial import of //depot/www/live/ from the state at revision #head
```

Our history became linear, just as though we had rebased before submitting (which is in fact exactly what happened). This means you can be free to create, work on, throw away, and merge branches on the Git side without fear that your history will somehow become incompatible with Perforce. If you can rebase it, you can contribute it to a Perforce server.

Branching

If your Perforce project has multiple branches, you're not out of luck; git-p4 can handle that in a way that makes it feel like Git. Let's say your Perforce depot is laid out like this:

```
//depot
  └── project
      ├── main
      └── dev
```

And let's say you have a dev branch, which has a view spec that looks like this:

```
//depot/project/main/... //depot/project/dev/...
```

Git-p4 can automatically detect that situation and do the right thing:

```
$ git p4 clone --detect-branches //depot/project@all
Importing from //depot/project@all into project
Initialized empty Git repository in /private/tmp/project/.git/
Importing revision 20 (50%)
  Importing new branch project/dev

  Resuming with change 20
  Importing revision 22 (100%)
  Updated branches: main dev
$ cd project; git log --oneline --all --graph --decorate
* eae77ae (HEAD, p4/master, p4/HEAD, master) main
| * 10d55fb (p4/project/dev) dev
| * a43cfaf Populate //depot/project/main/... //depot/project/dev/....
| /
* 2b83451 Project init
```

Note the “@all” specifier in the depot path; that tells git-p4 to clone not just the latest changeset for that subtree, but all changesets that have ever touched those paths. This is closer to Git’s concept of a clone, but if you’re working on a project with a long history, it could take a while.

The `--detect-branches` flag tells git-p4 to use Perforce’s branch specs to map the branches to Git refs. If these mappings aren’t present on the Perforce server (which is a perfectly valid way to use Perforce), you can tell git-p4 what the branch mappings are, and you get the same result:

```
$ git init project
Initialized empty Git repository in /tmp/project/.git/
$ cd project
$ git config git-p4.branchList main:dev
$ git clone --detect-branches //depot/project@all .
```

Setting the `git-p4.branchList` configuration variable to `main:dev` tells git-p4 that “main” and “dev” are both branches, and the second one is a child of the first one.

If we now `git checkout -b dev p4/project/dev` and make some commits, git-p4 is smart enough to target the right branch when we do `git p4 submit`. Unfortunately, git-p4 can’t mix shallow clones and multiple branches; if you have a huge project and want to work on more than one branch, you’ll have to `git p4 clone` once for each branch you want to submit to.

For creating or integrating branches, you’ll have to use a Perforce client. Git-p4 can only sync and submit to existing branches, and it can only do it one linear changeset at a time. If you merge two branches in Git and try to submit the new changeset, all that will be recorded is a bunch of file changes; the metadata about which branches are involved in the integration will be lost.

GIT AND PERFORCE SUMMARY

Git-p4 makes it possible to use a Git workflow with a Perforce server, and it’s pretty good at it. However, it’s important to remember that Perforce is in charge of the source, and you’re only using Git to work locally. Just be really careful about sharing Git commits; if you have a remote that other people use, don’t push any commits that haven’t already been submitted to the Perforce server.

If you want to freely mix the use of Perforce and Git as clients for source control, and you can convince the server administrator to install it, Git Fusion makes using Git a first-class version-control client for a Perforce server.

Git and TFS

Git is becoming popular with Windows developers, and if you're writing code on Windows, there's a good chance you're using Microsoft's Team Foundation Server (TFS). TFS is a collaboration suite that includes defect and work-item tracking, process support for Scrum and others, code review, and version control. There's a bit of confusion ahead: **TFS** is the server, which supports controlling source code using both Git and their own custom VCS, which they've dubbed **TFVC** (Team Foundation Version Control). Git support is a somewhat new feature for TFS (shipping with the 2013 version), so all of the tools that predate that refer to the version-control portion as "TFS", even though they're mostly working with TFVC.

If you find yourself on a team that's using TFVC but you'd rather use Git as your version-control client, there's a project for you.

WHICH TOOL

In fact, there are two: git-tf and git-tfs.

Git-tfs (found at <https://github.com/git-tfs/git-tfs>) is a .NET project, and (as of this writing) it only runs on Windows. To work with Git repositories, it uses the .NET bindings for libgit2, a library-oriented implementation of Git which is highly performant and allows a lot of flexibility with the guts of a Git repository. Libgit2 is not a complete implementation of Git, so to cover the difference git-tfs will actually call the command-line Git client for some operations, so there are no artificial limits on what it can do with Git repositories. Its support of TFVC features is very mature, since it uses the Visual Studio assemblies for operations with servers. This does mean you'll need access to those assemblies, which means you need to install a recent version of Visual Studio (any edition since version 2010, including Express since version 2012), or the Visual Studio SDK.

Git-tf (whose home is at <https://gittf.codeplex.com>) is a Java project, and as such runs on any computer with a Java runtime environment. It interfaces with Git repositories through JGit (a JVM implementation of Git), which means it has virtually no limitations in terms of Git functions. However, its support for TFVC is limited as compared to git-tfs – it does not support branches, for instance.

So each tool has pros and cons, and there are plenty of situations that favor one over the other. We'll cover the basic usage of both of them in this book.

You'll need access to a TFVC-based repository to follow along with these instructions. These aren't as plentiful in the wild as Git or Subversion repositories, so you may need to create one of your own. Codeplex (<https://www.codeplex.com>) or Visual Studio Online (<http://www.visualstudio.com>) are both good choices for this.

GETTING STARTED: GIT-TF

The first thing you do, just as with any Git project, is clone. Here's what that looks like with git-tf:

```
$ git tf clone https://tfs.codeplex.com:443/tfs/TFS13 $/myproject/Main project_git
```

The first argument is the URL of a TFVC collection, the second is of the form `$/project/branch`, and the third is the path to the local Git repository that is to be created (this last one is optional). Git-tf can only work with one branch at a time; if you want to make checkins on a different TFVC branch, you'll have to make a new clone from that branch.

This creates a fully functional Git repository:

```
$ cd project_git
$ git log --all --oneline --decorate
512e75a (HEAD, tag: TFS_C35190, origin_tfs/tfs, master) Checkin message
```

This is called a *shallow* clone, meaning that only the latest changeset has been downloaded. TFVC isn't designed for each client to have a full copy of the history, so git-tf defaults to only getting the latest version, which is much faster.

If you have some time, it's probably worth it to clone the entire project history, using the `--deep` option:

```
$ git tf clone https://tfs.codeplex.com:443/tfs/TFS13 $/myproject/Main \
  project_git --deep
Username: domain\user
Password:
Connecting to TFS...
Cloning $/myproject into /tmp/project_git: 100%, done.
Cloned 4 changesets. Cloned last changeset 35190 as d44b17a
$ cd project_git
$ git log --all --oneline --decorate
d44b17a (HEAD, tag: TFS_C35190, origin_tfs/tfs, master) Goodbye
126aa7b (tag: TFS_C35189)
8f77431 (tag: TFS_C35178) FIRST
```

```
0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the \
Team Project Creation Wizard
```

Notice the tags with names like TFS_C35189; this is a feature that helps you know which Git commits are associated with TFVC changesets. This is a nice way to represent it, since you can see with a simple log command which of your commits is associated with a snapshot that also exists in TFVC. They aren't necessary (and in fact you can turn them off with `git config git-tf.tag false`) – git-tf keeps the real commit-changeset mappings in the `.git/git-tf` file.

GETTING STARTED: GIT -TFS

Git-tfs cloning behaves a bit differently. Observe:

```
PS> git tfs clone --with-branches \
    https://username.visualstudio.com/DefaultCollection \
    $/project/Trunk project_git
Initialized empty Git repository in C:/Users/ben/project_git/.git/
C15 = b75da1aba1ffb359d00e85c52acb261e4586b0c9
C16 = c403405f4989d73a2c3c119e79021cb2104ce44a
Tfs branches found:
- $/tfvc-test/featureA
The name of the local branch will be : featureA
C17 = d202b53f67bde32171d5078968c644e562f1c439
C18 = 44cd729d8df868a8be20438fdeeffb961958b674
```

Notice the `--with-branches` flag. Git-tfs is capable of mapping TFVC branches to Git branches, and this flag tells it to set up a local Git branch for every TFVC branch. This is highly recommended if you've ever branched or merged in TFS, but it won't work with a server older than TFS 2010 – before that release, “branches” were just folders, so git-tfs can't tell them from regular folders.

Let's take a look at the resulting Git repository:

```
PS> git log --oneline --graph --decorate --all
* 44cd729 (tfvctest/featureA, featureA) Goodbye
* d202b53 Branched from $/tfvc-test/Trunk
* c403405 (HEAD, tfvctest/default, master) Hello
* b75da1a New project
PS> git log -1
commit c403405f4989d73a2c3c119e79021cb2104ce44a
Author: Ben Straub <ben@straub.cc>
Date:   Fri Aug 1 03:41:59 2014 +0000
```

Hello

```
git-tfs-id: [https://username.visualstudio.com/DefaultCollection]$/{myproject}/Trunk;C16
```

There are two local branches, `master` and `featureA`, which represent the initial starting point of the clone (Trunk in TFVC) and a child branch (`featureA` in TFVC). You can also see that the `tfss` “remote” has a couple of refs too: `default` and `featureA`, which represent TFVC branches. Git-tfs maps the branch you cloned from to `tfss/default`, and others get their own names.

Another thing to notice is the `git-tfs-id:` lines in the commit messages. Instead of tags, git-tfs uses these markers to relate TFVC changesets to Git commits. This has the implication that your Git commits will have a different SHA-1 hash before and after they have been pushed to TFVC.

GIT-TF[S] WORKFLOW

Regardless of which tool you’re using, you should set a couple of Git configuration values to avoid running into issues.

```
$ git config set --local core.ignorecase=true
$ git config set --local core.autocrlf=false
```

The obvious next thing you’re going to want to do is work on the project. TFVC and TFS have several features that may add complexity to your workflow:

1. Feature branches that aren’t represented in TFVC add a bit of complexity. This has to do with the **very** different ways that TFVC and Git represent branches.
2. Be aware that TFVC allows users to “checkout” files from the server, locking them so nobody else can edit them. This obviously won’t stop you from editing them in your local repository, but it could get in the way when it comes time to push your changes up to the TFVC server.
3. TFS has the concept of “gated” checkins, where a TFS build-test cycle has to complete successfully before the checkin is allowed. This uses the “shelve” function in TFVC, which we don’t cover in detail here. You can fake this in a manual fashion with `git-tf`, and `git-tfs` provides the `checkintool` command which is gate-aware.

In the interest of brevity, what we’ll cover here is the happy path, which side-steps or avoids most of these issues.

WORKFLOW: GIT-TF

Let's say you've done some work, made a couple of Git commits on `master`, and you're ready to share your progress on the TFVC server. Here's our Git repository:

```
$ git log --oneline --graph --decorate --all
* 4178a82 (HEAD, master) update code
* 9df2ae3 update readme
* d44b17a (tag: TFS_C35190, origin_tfs/tfs) Goodbye
* 126aa7b (tag: TFS_C35189)
* 8f77431 (tag: TFS_C35178) FIRST
* 0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the \
  Team Project Creation Wizard
```

We want to take the snapshot that's in the `4178a82` commit and push it up to the TFVC server. First things first: let's see if any of our teammates did anything since we last connected:

```
$ git tf fetch
Username: domain\user
Password:
Connecting to TFS...
Fetching $/myproject at latest changeset: 100%, done.
Downloaded changeset 35320 as commit 8ef06a8. Updated FETCH_HEAD.
$ git log --oneline --graph --decorate --all
* 8ef06a8 (tag: TFS_C35320, origin_tfs/tfs) just some text
| * 4178a82 (HEAD, master) update code
| * 9df2ae3 update readme
|/
* d44b17a (tag: TFS_C35190) Goodbye
* 126aa7b (tag: TFS_C35189)
* 8f77431 (tag: TFS_C35178) FIRST
* 0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the \
  Team Project Creation Wizard
```

Looks like someone else is working, too, and now we have divergent history. This is where Git shines, but we have two choices of how to proceed:

1. Making a merge commit feels natural as a Git user (after all, that's what `git pull` does), and `git-tf` can do this for you with a simple `git tf pull`. Be aware, however, that TFVC doesn't think this way, and if you push merge commits your history will start to look different on both

sides, which can be confusing. However, if you plan on submitting all of your changes as one changeset, this is probably the easiest choice.

2. Rebasing makes our commit history linear, which means we have the option of converting each of our Git commits into a TFVC changeset. Since this leaves the most options open, we recommend you do it this way; git-tf even makes it easy for you with `git tf pull --rebase`.

The choice is yours. For this example, we'll be rebasing:

```
$ git rebase FETCH_HEAD
First, rewinding head to replay your work on top of it...
Applying: update readme
Applying: update code
$ git log --oneline --graph --decorate --all
* 5a0e25e (HEAD, master) update code
* 6eb3eb5 update readme
* 8ef06a8 (tag: TFS_C35320, origin_tfs/tfs) just some text
* d44b17a (tag: TFS_C35190) Goodbye
* 126aa7b (tag: TFS_C35189)
* 8f77431 (tag: TFS_C35178) FIRST
* 0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the \
    Team Project Creation Wizard
```

Now we're ready to make a checkin to the TFVC server. Git-tf gives you the choice of making a single changeset that represents all the changes since the last one (`--shallow`, which is the default) and creating a new changeset for each Git commit (`--deep`). For this example, we'll just create one changeset:

```
$ git tf checkin -m 'Updating readme and code'
Username: domain\user
Password:
Connecting to TFS...
Checking in to $/myproject: 100%, done.
Checked commit 5a0e25e in as changeset 35348
$ git log --oneline --graph --decorate --all
* 5a0e25e (HEAD, tag: TFS_C35348, origin_tfs/tfs, master) update code
* 6eb3eb5 update readme
* 8ef06a8 (tag: TFS_C35320) just some text
* d44b17a (tag: TFS_C35190) Goodbye
* 126aa7b (tag: TFS_C35189)
* 8f77431 (tag: TFS_C35178) FIRST
* 0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the \
    Team Project Creation Wizard
```

There's a new TFS_C35348 tag, indicating that TFVC is storing the exact same snapshot as the 5a0e25e commit. It's important to note that not every Git commit needs to have an exact counterpart in TFVC; the 6eb3eb5 commit, for example, doesn't exist anywhere on the server.

That's the main workflow. There are a couple of other considerations you'll want to keep in mind:

- There is no branching. Git-tf can only create Git repositories from one TFVC branch at a time.
- Collaborate using either TFVC or Git, but not both. Different git-tf clones of the same TFVC repository may have different commit SHA-1 hashes, which will cause no end of headaches.
- If your team's workflow includes collaborating in Git and syncing periodically with TFVC, only connect to TFVC with one of the Git repositories.

WORKFLOW: GIT-TFS

Let's walk through the same scenario using git-tfs. Here are the new commits we've made to the master branch in our Git repository:

```
PS> git log --oneline --graph --all --decorate
* c3bd3ae (HEAD, master) update code
* d85e5a2 update readme
| * 44cd729 (tfv/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|/
* c403405 (tfv/default) Hello
* b75da1a New project
```

Now let's see if anyone else has done work while we were hacking away:

```
PS> git tfs fetch
C19 = aea74a0313de0a391940c999e51c5c15c381d91d
PS> git log --all --oneline --graph --decorate
* aea74a0 (tfv/default) update documentation
| * c3bd3ae (HEAD, master) update code
| * d85e5a2 update readme
|/
| * 44cd729 (tfv/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|/
* c403405 Hello
* b75da1a New project
```

Yes, it turns out our coworker has added a new TFVC changeset, which shows up as the new `aea74a0` commit, and the `tfv/default` remote branch has moved.

As with `git-tf`, we have two fundamental options for how to resolve this divergent history:

1. Rebase to preserve a linear history.
2. Merge to preserve what actually happened.

In this case, we're going to do a "deep" checkin, where every Git commit becomes a TFVC changeset, so we want to rebase.

```
PS> git rebase tfv/default
First, rewinding head to replay your work on top of it...
Applying: update readme
Applying: update code
PS> git log --all --oneline --graph --decorate
* 10a75ac (HEAD, master) update code
* 5cec4ab update readme
* aea74a0 (tfv/default) update documentation
| * 44cd729 (tfv/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|/
* c403405 Hello
* b75da1a New project
```

Now we're ready to complete our contribution by checking in our code to the TFVC server. We'll use the `rcheckin` command here to create a TFVC changeset for each Git commit in the path from HEAD to the first `tfv` remote branch found (the `checkin` command would only create one changeset, sort of like squashing Git commits).

```
PS> git tfv rcheckin
Working with tfv remote: default
Fetching changes from TFS to minimize possibility of late conflict...
Starting checkin of 5cec4ab4 'update readme'
  add README.md
C20 = 71a5ddce274c19f8fdc322b4f165d93d89121017
Done with 5cec4ab4b213c354341f66c80cd650ab98dcf1ed, rebasing tail onto new TFS-commit...
Rebase done successfully.
Starting checkin of b1bf0f99 'update code'
  edit .git\tfv\default\workspace\ConsoleApplication1\ConsoleApplication1\Program.cs
C21 = ff04e7c35dfbe6a8f94e782bf5e0031cee8d103b
Done with b1bf0f9977b2d48bad611ed4a03d3738df05ea5d, rebasing tail onto new TFS-commit...
Rebase done successfully.
No more to rcheckin.
PS> git log --all --oneline --graph --decorate
```

```
* ff04e7c (HEAD, tfs/default, master) update code
* 71a5ddc update readme
*aea74a0 update documentation
| * 44cd729 (tfs/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|/
* c403405 Hello
* b75da1a New project
```

Notice how after every successful checkin to the TFVC server, git-tfs is rebasing the remaining work onto what it just did. That's because it's adding the git-tfs-id field to the bottom of the commit messages, which changes the SHA-1 hashes. This is exactly as designed, and there's nothing to worry about, but you should be aware that it's happening, especially if you're sharing Git commits with others.

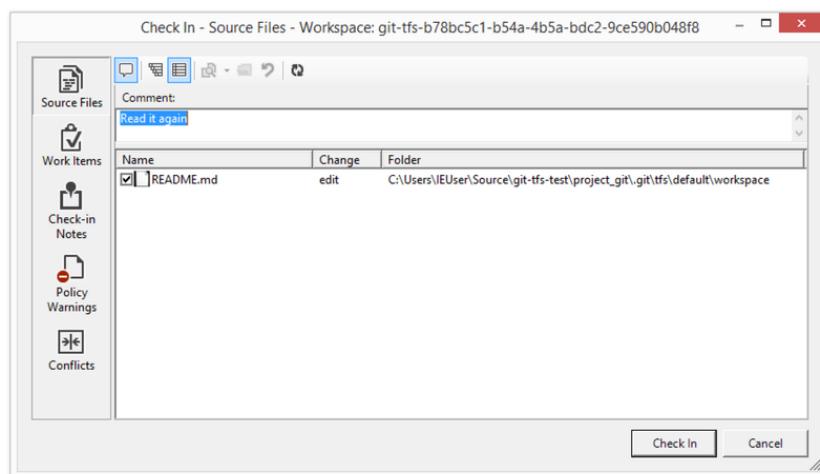
TFS has many features that integrate with its version control system, such as work items, designated reviewers, gated checkins, and so on. It can be cumbersome to work with these features using only a command-line tool, but fortunately git-tfs lets you launch a graphical checkin tool very easily:

```
PS> git tfs checkintool
PS> git tfs ct
```

It looks a bit like this:

FIGURE 9-3

The *git-tfs* checkin tool.



This will look familiar to TFS users, as it's the same dialog that's launched from within Visual Studio.

Git-tfs also lets you control TFVC branches from your Git repository. As an example, let's create one:

```
PS> git tfs branch $/tfvc-test/featureBee
The name of the local branch will be : featureBee
C26 = 1d54865c397608c004a2cadce7296f5edc22a7e5
PS> git log --oneline --graph --decorate --all
* 1d54865 (tfv/featureBee) Creation branch $/myproject/featureBee
* ff04e7c (HEAD, tfs/default, master) update code
* 71a5ddc update readme
* aea74a0 update documentation
| * 44cd729 (tfv/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|/
* c403405 Hello
* b75da1a New project
```

Creating a branch in TFVC means adding a changeset where that branch now exists, and this is projected as a Git commit. Note also that git-tfs **created** the `tfv/featureBee` remote branch, but HEAD is still pointing to `master`. If you want to work on the newly-minted branch, you'll want to base your new commits on the `1d54865` commit, perhaps by creating a topic branch from that commit.

GIT AND TFS SUMMARY

Git-tf and Git-tfs are both great tools for interfacing with a TFVC server. They allow you to use the power of Git locally, avoid constantly having to round-trip to the central TFVC server, and make your life as a developer much easier, without forcing your entire team to migrate to Git. If you're working on Windows (which is likely if your team is using TFS), you'll probably want to use git-tfs, since its feature set is more complete, but if you're working on another platform, you'll be using git-tf, which is more limited. As with most of the tools in this chapter, you should choose one of these version-control systems to be canonical, and use the other one in a subordinate fashion – either Git or TFVC should be the center of collaboration, but not both.

Migrating to Git

If you have an existing codebase in another VCS but you've decided to start using Git, you must migrate your project one way or another. This section goes

over some importers for common systems, and then demonstrates how to develop your own custom importer. You'll learn how to import data from several of the bigger professionally used SCM systems, because they make up the majority of users who are switching, and because high-quality tools for them are easy to come by.

Subversion

If you read the previous section about using `git svn`, you can easily use those instructions to `git svn clone` a repository; then, stop using the Subversion server, push to a new Git server, and start using that. If you want the history, you can accomplish that as quickly as you can pull the data out of the Subversion server (which may take a while).

However, the import isn't perfect; and because it will take so long, you may as well do it right. The first problem is the author information. In Subversion, each person committing has a user on the system who is recorded in the commit information. The examples in the previous section show `schacon` in some places, such as the `blame` output and the `git svn log`. If you want to map this to better Git author data, you need a mapping from the Subversion users to the Git authors. Create a file called `users.txt` that has this mapping in a format like this:

```
schacon = Scott Chacon <schacon@geemail.com>
selse = Someo Nelse <selse@geemail.com>
```

To get a list of the author names that SVN uses, you can run this:

```
$ svn log --xml | grep author | sort -u | \
perl -pe 's/.*/>(.*)<.*/$1 = /'
```

That generates the log output in XML format, then keeps only the lines with author information, discards duplicates, strips out the XML tags. (Obviously this only works on a machine with `grep`, `sort`, and `perl` installed.) Then, redirect that output into your `users.txt` file so you can add the equivalent Git user data next to each entry.

You can provide this file to `git svn` to help it map the author data more accurately. You can also tell `git svn` not to include the metadata that Subversion normally imports, by passing `--no-metadata` to the `clone` or `init` command. This makes your `import` command look like this:

```
$ git svn clone http://my-project.googlecode.com/svn/ \
--authors-file=users.txt --no-metadata -s my_project
```

Now you should have a nicer Subversion import in your `my_project` directory. Instead of commits that look like this

```
commit 37efa680e8473b615de980fa935944215428a35a
Author: schacon <schacon@4c93b258-373f-11de-be05-5f7a86268029>
Date:   Sun May 3 00:12:22 2009 +0000

    fixed install - go to trunk

git-svn-id: https://my-project.googlecode.com/svn/trunk@94 4c93b258-373f-11de-
be05-5f7a86268029
```

they look like this:

```
commit 03a8785f44c8ea5cdb0e8834b7c8e6c469be2ff2
Author: Scott Chacon <schacon@geemail.com>
Date:   Sun May 3 00:12:22 2009 +0000

    fixed install - go to trunk
```

Not only does the Author field look a lot better, but the `git-svn-id` is no longer there, either.

You should also do a bit of post-import cleanup. For one thing, you should clean up the weird references that `git svn` set up. First you'll move the tags so they're actual tags rather than strange remote branches, and then you'll move the rest of the branches so they're local.

To move the tags to be proper Git tags, run

```
$ cp -Rf .git/refs/remotes/origin/tags/* .git/refs/tags/
$ rm -Rf .git/refs/remotes/origin/tags
```

This takes the references that were remote branches that started with `remotes/origin/tags/` and makes them real (lightweight) tags.

Next, move the rest of the references under `refs/remotes` to be local branches:

```
$ cp -Rf .git/refs/remotes/* .git/refs/heads/
$ rm -Rf .git/refs/remotes
```

Now all the old branches are real Git branches and all the old tags are real Git tags. The last thing to do is add your new Git server as a remote and push to it. Here is an example of adding your server as a remote:

```
$ git remote add origin git@my-git-server:myrepository.git
```

Because you want all your branches and tags to go up, you can now run this:

```
$ git push origin --all
```

All your branches and tags should be on your new Git server in a nice, clean import.

Mercurial

Since Mercurial and Git have fairly similar models for representing versions, and since Git is a bit more flexible, converting a repository from Mercurial to Git is fairly straightforward, using a tool called “hg-fast-export”, which you’ll need a copy of:

```
$ git clone http://repo.or.cz/r/fast-export.git /tmp/fast-export
```

The first step in the conversion is to get a full clone of the Mercurial repository you want to convert:

```
$ hg clone <remote repo URL> /tmp/hg-repo
```

The next step is to create an author mapping file. Mercurial is a bit more forgiving than Git for what it will put in the author field for changesets, so this is a good time to clean house. Generating this is a one-line command in a bash shell:

```
$ cd /tmp/hg-repo
$ hg log | grep user: | sort | uniq | sed 's/user: *//' > ../authors
```

This will take a few seconds, depending on how long your project’s history is, and afterwards the `/tmp/authors` file will look something like this:

```

bob
bob@localhost
bob <bob@company.com>
bob jones <bob <AT> company <DOT> com>
Bob Jones <bob@company.com>
Joe Smith <joe@company.com>
```

In this example, the same person (Bob) has created changesets under four different names, one of which actually looks correct, and one of which would be completely invalid for a Git commit. Hg-fast-export lets us fix this by adding ={new name and email address} at the end of every line we want to change, and removing the lines for any usernames that we want to leave alone. If all the usernames look fine, we won't need this file at all. In this example, we want our file to look like this:

```

bob=Bob Jones <bob@company.com>
bob@localhost=Bob Jones <bob@company.com>
bob jones <bob <AT> company <DOT> com>=Bob Jones <bob@company.com>
bob <bob@company.com>=Bob Jones <bob@company.com>
```

The next step is to create our new Git repository, and run the export script:

```
$ git init /tmp/converted
$ cd /tmp/converted
$ /tmp/fast-export/hg-fast-export.sh -r /tmp/hg-repo -A /tmp/authors
```

The -r flag tells hg-fast-export where to find the Mercurial repository we want to convert, and the -A flag tells it where to find the author-mapping file. The script parses Mercurial changesets and converts them into a script for Git's "fast-import" feature (which we'll discuss in detail a bit later on). This takes a bit (though it's *much* faster than it would be over the network), and the output is fairly verbose:

```
$ /tmp/fast-export/hg-fast-export.sh -r /tmp/hg-repo -A /tmp/authors
Loaded 4 authors
master: Exporting full revision 1/22208 with 13/0/0 added/changed/removed files
master: Exporting simple delta revision 2/22208 with 1/1/0 added/changed/removed files
master: Exporting simple delta revision 3/22208 with 0/1/0 added/changed/removed files
[...]
master: Exporting simple delta revision 22206/22208 with 0/4/0 added/changed/removed files
master: Exporting simple delta revision 22207/22208 with 0/2/0 added/changed/removed files
master: Exporting thorough delta revision 22208/22208 with 3/213/0 added/changed/removed files
Exporting tag [0.4c] at [hg r9] [git :10]
Exporting tag [0.4d] at [hg r16] [git :17]
```

```
[...]
Exporting tag [3.1-rc] at [hg r21926] [git :21927]
Exporting tag [3.1] at [hg r21973] [git :21974]
Issued 22315 commands
git-fast-import statistics:
-----
Alloc'd objects:      120000
Total objects:      115032 (      208171 duplicates )          39602
    blobs :        40504 (      205320 duplicates )          26117 deltas of 47599
    trees :        52320 (       2851 duplicates )          47467 deltas of 0
    commits:        22208 (           0 duplicates )          0 deltas of 0
    tags :          0 (           0 duplicates )          0 deltas of 0
Total branches:      109 (         2 loads      )
marks:            1048576 (     22208 unique      )
atoms:             1952
Memory total:      7860 KiB
    pools:          2235 KiB
objects:          5625 KiB
-----
pack_report: getpagesize() =      4096
pack_report: core.packedGitWindowSize = 1073741824
pack_report: core.packedGitLimit = 8589934592
pack_report: pack_used_ctr =      90430
pack_report: pack_mmap_calls =      46771
pack_report: pack_open_windows =      1 /      1
pack_report: pack_mapped = 340852700 / 340852700
-----
$ git shortlog -sn
 369 Bob Jones
 365 Joe Smith
```

That's pretty much all there is to it. All of the Mercurial tags have been converted to Git tags, and Mercurial branches and bookmarks have been converted to Git branches. Now you're ready to push the repository up to its new server-side home:

```
$ git remote add origin git@my-git-server:myrepository.git  
$ git push origin --all
```

Perforce

The next system you'll look at importing from is Perforce. As we discussed above, there are two ways to let Git and Perforce talk to each other: git-p4 and Perforce Git Fusion.

PERFORCE GIT FUSION

Git Fusion makes this process fairly painless. Just configure your project settings, user mappings, and branches using a configuration file (as discussed in “[Git Fusion](#)”), and clone the repository. Git Fusion leaves you with what looks like a native Git repository, which is then ready to push to a native Git host if you desire. You could even use Perforce as your Git host if you like.

GIT-P4

Git-p4 can also act as an import tool. As an example, we’ll import the Jam project from the Perforce Public Depot. To set up your client, you must export the P4PORT environment variable to point to the Perforce depot:

```
$ export P4PORT=public.perforce.com:1666
```

In order to follow along, you’ll need a Perforce depot to connect with. We’ll be using the public depot at public.perforce.com for our examples, but you can use any depot you have access to.

Run the `git p4 clone` command to import the Jam project from the Perforce server, supplying the depot and project path and the path into which you want to import the project:

```
$ git-p4 clone //guest/perforce_software/jam@all p4import
Importing from //guest/perforce_software/jam@all into p4import
Initialized empty Git repository in /private/tmp/p4import/.git/
Import destination: refs/remotes/p4/master
Importing revision 9957 (100%)
```

This particular project has only one branch, but if you have branches that are configured with branch views (or just a set of directories), you can use the `--detect-branches` flag to `git p4 clone` to import all the project’s branches as well. See “[Branching](#)” for a bit more detail on this.

At this point you’re almost done. If you go to the `p4import` directory and run `git log`, you can see your imported work:

```
$ git log -2
commit e5da1c909e5db3036475419f6379f2c73710c4e6
Author: giles <giles@giles@perforce.com>
Date:   Wed Feb 8 03:13:27 2012 -0800
```

```
Correction to line 355; change </UL> to </OL>.

[git-p4: depot-paths = "//public/jam/src/": change = 8068]

commit aa21359a0a135dda85c50a7f7cf249e4f7b8fd98
Author: kwirth <kwirth@perforce.com>
Date:   Tue Jul 7 01:35:51 2009 -0800

    Fix spelling error on Jam doc page (cummulative -> cumulative).

[git-p4: depot-paths = "//public/jam/src/": change = 7304]
```

You can see that `git-p4` has left an identifier in each commit message. It's fine to keep that identifier there, in case you need to reference the Perforce change number later. However, if you'd like to remove the identifier, now is the time to do so – before you start doing work on the new repository. You can use `git filter-branch` to remove the identifier strings en masse:

```
$ git filter-branch --msg-filter 'sed -e "/^\\[git-p4:/d"''
Rewrite e5da1c909e5db3036475419f6379f2c73710c4e6 (125/125)
Ref 'refs/heads/master' was rewritten
```

If you run `git log`, you can see that all the SHA-1 checksums for the commits have changed, but the `git-p4` strings are no longer in the commit messages:

```
$ git log -2
commit b17341801ed838d97f7800a54a6f9b95750839b7
Author: giles <giles@giles@perforce.com>
Date:   Wed Feb 8 03:13:27 2012 -0800

    Correction to line 355; change </UL> to </OL>.

commit 3e68c2e26cd89cb983eb52c024ecdfba1d6b3fff
Author: kwirth <kwirth@perforce.com>
Date:   Tue Jul 7 01:35:51 2009 -0800

    Fix spelling error on Jam doc page (cummulative -> cumulative).
```

Your import is ready to push up to your new Git server.

TFS

If your team is converting their source control from TFVC to Git, you'll want the highest-fidelity conversion you can get. This means that, while we covered both git-tfs and git-tf for the interop section, we'll only be covering git-tfs for this part, because git-tfs supports branches, and this is prohibitively difficult using git-tf.

This is a one-way conversion. The resulting Git repository won't be able to connect with the original TFVC project.

The first thing to do is map usernames. TFVC is fairly liberal with what goes into the author field for changesets, but Git wants a human-readable name and email address. You can get this information from the `tf` command-line client, like so:

```
PS> tf history $/myproject -recursive > AUTHORS_TMP
```

This grabs all of the changesets in the history of the project and put it in the `AUTHORS_TMP` file that we will process to extract the data of the *User* column (the 2nd one). Open the file and find at which characters start and end the column and replace, in the following command-line, the parameters `11-20` of the `cut` command with the ones found:

```
PS> cat AUTHORS_TMP | cut -b 11-20 | tail -n+3 | uniq | sort > AUTHORS
```

The `cut` command keeps only the characters between 11 and 20 from each line. The `tail` command skips the first two lines, which are field headers and ASCII-art underlines. The result of all of this is piped to `uniq` to eliminate duplicates, and saved to a file named `AUTHORS`. The next step is manual; in order for git-tfs to make effective use of this file, each line must be in this format:

```
DOMAIN\username = User Name <email@address.com>
```

The portion on the left is the “User” field from TFVC, and the portion on the right side of the equals sign is the user name that will be used for Git commits.

Once you have this file, the next thing to do is make a full clone of the TFVC project you're interested in:

```
PS> git tfs clone --with-branches --authors=AUTHORS https://username.visualstudio.com/Default
```

Next you'll want to clean the `git-tfs-id` sections from the bottom of the commit messages. The following command will do that:

```
PS> git filter-branch -f --msg-filter 'sed "s/^git-tfs-id:.*$/g"' -- --all
```

That uses the `sed` command from the Git-bash environment to replace any line starting with “`git-tfs-id:`” with emptiness, which Git will then ignore.

Once that’s all done, you’re ready to add a new remote, push all your branches up, and have your team start working from Git.

A Custom Importer

If your system isn’t one of the above, you should look for an importer online – quality importers are available for many other systems, including CVS, Clear Case, Visual Source Safe, even a directory of archives. If none of these tools works for you, you have a more obscure tool, or you otherwise need a more custom importing process, you should use `git fast-import`. This command reads simple instructions from `stdin` to write specific Git data. It’s much easier to create Git objects this way than to run the raw Git commands or try to write the raw objects (see [Chapter 10](#) for more information). This way, you can write an import script that reads the necessary information out of the system you’re importing from and prints straightforward instructions to `stdout`. You can then run this program and pipe its output through `git fast-import`.

To quickly demonstrate, you’ll write a simple importer. Suppose you work in `current`, you back up your project by occasionally copying the directory into a time-stamped `back_YYYY_MM_DD` backup directory, and you want to import this into Git. Your directory structure looks like this:

```
$ ls /opt/import_from
back_2014_01_02
back_2014_01_04
back_2014_01_14
back_2014_02_03
current
```

In order to import a Git directory, you need to review how Git stores its data. As you may remember, Git is fundamentally a linked list of commit objects that point to a snapshot of content. All you have to do is tell `fast-import` what the content snapshots are, what commit data points to them, and the order they go in. Your strategy will be to go through the snapshots one at a time and create commits with the contents of each directory, linking each commit back to the previous one.

As we did in “[Un ejemplo de implantación de una determinada política en Git](#)”, we’ll write this in Ruby, because it’s what we generally work with and it

tends to be easy to read. You can write this example pretty easily in anything you're familiar with – it just needs to print the appropriate information to `stdout`. And, if you are running on Windows, this means you'll need to take special care to not introduce carriage returns at the end your lines – git fast-import is very particular about just wanting line feeds (LF) not the carriage return line feeds (CRLF) that Windows uses.

To begin, you'll change into the target directory and identify every subdirectory, each of which is a snapshot that you want to import as a commit. You'll change into each subdirectory and print the commands necessary to export it. Your basic main loop looks like this:

```
last_mark = nil

# loop through the directories
Dir.chdir(argv[0]) do
  Dir.glob("*").each do |dir|
    next if File.file?(dir)

    # move into the target directory
    Dir.chdir(dir) do
      last_mark = print_export(dir, last_mark)
    end
  end
end
```

You run `print_export` inside each directory, which takes the manifest and mark of the previous snapshot and returns the manifest and mark of this one; that way, you can link them properly. “Mark” is the `fast-import` term for an identifier you give to a commit; as you create commits, you give each one a mark that you can use to link to it from other commits. So, the first thing to do in your `print_export` method is generate a mark from the directory name:

```
mark = convert_dir_to_mark(dir)
```

You'll do this by creating an array of directories and using the index value as the mark, because a mark must be an integer. Your method looks like this:

```
$marks = []
def convert_dir_to_mark(dir)
  if !$marks.include?(dir)
    $marks << dir
  end
  ($marks.index(dir) + 1).to_s
end
```

Now that you have an integer representation of your commit, you need a date for the commit metadata. Because the date is expressed in the name of the directory, you'll parse it out. The next line in your `print_export` file is

```
date = convert_dir_to_date(dir)
```

where `convert_dir_to_date` is defined as

```
def convert_dir_to_date(dir)
  if dir == 'current'
    return Time.now().to_i
  else
    dir = dir.gsub('back_', '')
    (year, month, day) = dir.split('_')
    return Time.local(year, month, day).to_i
  end
end
```

That returns an integer value for the date of each directory. The last piece of meta-information you need for each commit is the committer data, which you hardcode in a global variable:

```
$author = 'John Doe <john@example.com>'
```

Now you're ready to begin printing out the commit data for your importer. The initial information states that you're defining a commit object and what branch it's on, followed by the mark you've generated, the committer information and commit message, and then the previous commit, if any. The code looks like this:

```
# print the import information
puts 'commit refs/heads/master'
puts 'mark :' + mark
puts "committer #{$author} #{date} -0700"
export_data('imported from ' + dir)
puts 'from :' + last_mark if last_mark
```

You hardcode the time zone (-0700) because doing so is easy. If you're importing from another system, you must specify the time zone as an offset. The commit message must be expressed in a special format:

```
data (size)\n(contents)
```

The format consists of the word `data`, the size of the data to be read, a new-line, and finally the data. Because you need to use the same format to specify the file contents later, you create a helper method, `export_data`:

```
def export_data(string)
  print "data #{string.size}\n#{string}"
end
```

All that's left is to specify the file contents for each snapshot. This is easy, because you have each one in a directory – you can print out the `deleteall` command followed by the contents of each file in the directory. Git will then record each snapshot appropriately:

```
puts 'deleteall'
Dir.glob("**/*").each do |file|
  next if !File.file?(file)
  inline_data(file)
end
```

Note: Because many systems think of their revisions as changes from one commit to another, `fast-import` can also take commands with each commit to specify which files have been added, removed, or modified and what the new contents are. You could calculate the differences between snapshots and provide only this data, but doing so is more complex – you may as well give Git all the data and let it figure it out. If this is better suited to your data, check the `fast-import` man page for details about how to provide your data in this manner.

The format for listing the new file contents or specifying a modified file with the new contents is as follows:

```
M 644 inline path/to/file
data (size)
(file contents)
```

Here, 644 is the mode (if you have executable files, you need to detect and specify 755 instead), and `inline` says you'll list the contents immediately after this line. Your `inline_data` method looks like this:

```
def inline_data(file, code = 'M', mode = '644')
  content = File.read(file)
  puts "#{code} #{mode} inline #{file}"
  export_data(content)
end
```

You reuse the `export_data` method you defined earlier, because it's the same as the way you specified your commit message data.

The last thing you need to do is to return the current mark so it can be passed to the next iteration:

```
return mark
```

If you are running on Windows you'll need to make sure that you add one extra step. As mentioned before, Windows uses CRLF for new line characters while git fast-import expects only LF. To get around this problem and make git fast-import happy, you need to tell ruby to use LF instead of CRLF:

```
$stdout.binmode
```

That's it. Here's the script in its entirety:

```
#!/usr/bin/env ruby

$stdout.binmode
$author = "John Doe <john@example.com>"

$marks = []
def convert_dir_to_mark(dir)
  if !$marks.include?(dir)
    $marks << dir
  end
  ($marks.index(dir)+1).to_s
end

def convert_dir_to_date(dir)
  if dir == 'current'
    return Time.now().to_i
  else
    dir = dir.gsub('back_', '')
    (year, month, day) = dir.split('_')
    return Time.local(year, month, day).to_i
  end
end

def export_data(string)
  print "data #{string.size}\n#{string}"
end

def inline_data(file, code='M', mode='644')
  content = File.read(file)
  puts "#{code} #{mode} inline #{file}"
```

```

        export_data(content)
end

def print_export(dir, last_mark)
  date = convert_dir_to_date(dir)
  mark = convert_dir_to_mark(dir)

  puts 'commit refs/heads/master'
  puts "mark :#{mark}"
  puts "committer #{author} #{date} -0700"
  export_data("imported from #{dir}")
  puts "from :#[last_mark]" if last_mark

  puts 'deleteall'
  Dir.glob("**/*").each do |file|
    next if !File.file?(file)
    inline_data(file)
  end
  mark
end

# Loop through the directories
last_mark = nil
Dir.chdir(argv[0]) do
  Dir.glob("*").each do |dir|
    next if File.file?(dir)

    # move into the target directory
    Dir.chdir(dir) do
      last_mark = print_export(dir, last_mark)
    end
  end
end

```

If you run this script, you'll get content that looks something like this:

```
$ ruby import.rb /opt/import_from
commit refs/heads/master
mark :1
committer John Doe <john@example.com> 1388649600 -0700
data 29
imported from back_2014_01_02deleteall
M 644 inline README.md
data 28
# Hello
```

```
This is my readme.
commit refs/heads/master
```

```

mark :2
committer John Doe <john@example.com> 1388822400 -0700
data 29
imported from back_2014_01_04from :1
deleteall
M 644 inline main.rb
data 34
#!/bin/env ruby

puts "Hey there"
M 644 inline README.md
(...)

```

To run the importer, pipe this output through `git fast-import` while in the Git directory you want to import into. You can create a new directory and then run `git init` in it for a starting point, and then run your script:

```

$ git init
Initialized empty Git repository in /opt/import_to/.git/
$ ruby import.rb /opt/import_from | git fast-import
git-fast-import statistics:
-----
Alloc'd objects:      5000
Total objects:       13 (      6 duplicates ) )
          blobs :      5 (      4 duplicates ) 3 deltas of
          trees :      4 (      1 duplicates ) 0 deltas of
          commits:      4 (      1 duplicates ) 0 deltas of
          tags :      0 (      0 duplicates ) 0 deltas of
Total branches:       1 (      1 loads    )
          marks:      1024 (      5 unique   )
          atoms:        2
Memory total:        2344 KiB
          pools:        2110 KiB
          objects:       234 KiB
-----
pack_report: getpagesize() = 4096
pack_report: core.packedGitWindowSize = 1073741824
pack_report: core.packedGitLimit = 8589934592
pack_report: pack_used_ctr = 10
pack_report: pack_mmap_calls = 5
pack_report: pack_open_windows = 2 / 2
pack_report: pack_mapped = 1457 / 1457
-----
```

As you can see, when it completes successfully, it gives you a bunch of statistics about what it accomplished. In this case, you imported 13 objects total for 4 commits into 1 branch. Now, you can run `git log` to see your new history:

```
$ git log -2
commit 3caa046d4aac682a55867132ccdfbe0d3fdee498
Author: John Doe <john@example.com>
Date:   Tue Jul 29 19:39:04 2014 -0700

        imported from current

commit 4afc2b945d0d3c8cd00556fbe2e8224569dc9def
Author: John Doe <john@example.com>
Date:   Mon Feb 3 01:00:00 2014 -0700

        imported from back_2014_02_03
```

There you go – a nice, clean Git repository. It’s important to note that nothing is checked out – you don’t have any files in your working directory at first. To get them, you must reset your branch to where `master` is now:

```
$ ls
$ git reset --hard master
HEAD is now at 3caa046 imported from current
$ ls
README.md main.rb
```

You can do a lot more with the `fast-import` tool – handle different modes, binary data, multiple branches and merging, tags, progress indicators, and more. A number of examples of more complex scenarios are available in the `contrib/fast-import` directory of the Git source code.

Summary

You should feel comfortable using Git as a client for other version-control systems, or importing nearly any existing repository into Git without losing data. In the next chapter, we’ll cover the raw internals of Git so you can craft every single byte, if need be.

Git Internals 10

You may have skipped to this chapter from a previous chapter, or you may have gotten here after reading the rest of the book – in either case, this is where we'll go over the inner workings and implementation of Git. We found that learning this information was fundamentally important to understanding how useful and powerful Git is, but others have argued to us that it can be confusing and unnecessarily complex for beginners. Thus, we've made this discussion the last chapter in the book so you could read it early or later in your learning process. We leave it up to you to decide.

Now that you're here, let's get started. First, if it isn't yet clear, Git is fundamentally a content-addressable filesystem with a VCS user interface written on top of it. You'll learn more about what this means in a bit.

In the early days of Git (mostly pre 1.5), the user interface was much more complex because it emphasized this filesystem rather than a polished VCS. In the last few years, the UI has been refined until it's as clean and easy to use as any system out there; but often, the stereotype lingers about the early Git UI that was complex and difficult to learn.

The content-addressable filesystem layer is amazingly cool, so we'll cover that first in this chapter; then, you'll learn about the transport mechanisms and the repository maintenance tasks that you may eventually have to deal with.

Plumbing and Porcelain

This book covers how to use Git with 30 or so verbs such as `checkout`, `branch`, `remote`, and so on. But because Git was initially a toolkit for a VCS rather than a full user-friendly VCS, it has a bunch of verbs that do low-level work and were designed to be chained together UNIX style or called from scripts. These commands are generally referred to as “plumbing” commands, and the more user-friendly commands are called “porcelain” commands.

The book's first nine chapters deal almost exclusively with porcelain commands. But in this chapter, you'll be dealing mostly with the lower-level plumb-

ing commands, because they give you access to the inner workings of Git, and help demonstrate how and why Git does what it does. Many of these commands aren't meant to be used manually on the command line, but rather to be used as building blocks for new tools and custom scripts.

When you run `git init` in a new or existing directory, Git creates the `.git` directory, which is where almost everything that Git stores and manipulates is located. If you want to back up or clone your repository, copying this single directory elsewhere gives you nearly everything you need. This entire chapter basically deals with the stuff in this directory. Here's what it looks like:

```
$ ls -F1
HEAD
config*
description
hooks/
info/
objects/
refs/
```

You may see some other files in there, but this is a fresh `git init` repository – it's what you see by default. The `description` file is only used by the Git-Web program, so don't worry about it. The `config` file contains your project-specific configuration options, and the `info` directory keeps a global exclude file for ignored patterns that you don't want to track in a `.gitignore` file. The `hooks` directory contains your client- or server-side hook scripts, which are discussed in detail in “[Puntos de enganche en Git](#)”.

This leaves four important entries: the `HEAD` and (yet to be created) `index` files, and the `objects` and `refs` directories. These are the core parts of Git. The `objects` directory stores all the content for your database, the `refs` directory stores pointers into commit objects in that data (branches), the `HEAD` file points to the branch you currently have checked out, and the `index` file is where Git stores your staging area information. You'll now look at each of these sections in detail to see how Git operates.

Git Objects

Git is a content-addressable filesystem. Great. What does that mean? It means that at the core of Git is a simple key-value data store. You can insert any kind of content into it, and it will give you back a key that you can use to retrieve the content again at any time. To demonstrate, you can use the plumbing command `hash-object`, which takes some data, stores it in your `.git` directory,

and gives you back the key the data is stored as. First, you initialize a new Git repository and verify that there is nothing in the `objects` directory:

```
$ git init test
Initialized empty Git repository in /tmp/test/.git/
$ cd test
$ find .git/objects
.git/objects
.git/objects/info
.git/objects/pack
$ find .git/objects -type f
```

Git has initialized the `objects` directory and created `pack` and `info` subdirectories in it, but there are no regular files. Now, store some text in your Git database:

```
$ echo 'test content' | git hash-object -w --stdin
d670460b4b4aece5915caf5c68d12f560a9fe3e4
```

The `-w` tells `hash-object` to store the object; otherwise, the command simply tells you what the key would be. `--stdin` tells the command to read the content from `stdin`; if you don't specify this, `hash-object` expects a file path at the end. The output from the command is a 40-character checksum hash. This is the SHA-1 hash – a checksum of the content you're storing plus a header, which you'll learn about in a bit. Now you can see how Git has stored your data:

```
$ find .git/objects -type f
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

You can see a file in the `objects` directory. This is how Git stores the content initially – as a single file per piece of content, named with the SHA-1 checksum of the content and its header. The subdirectory is named with the first 2 characters of the SHA-1, and the filename is the remaining 38 characters.

You can pull the content back out of Git with the `cat-file` command. This command is sort of a Swiss army knife for inspecting Git objects. Passing `-p` to it instructs the `cat-file` command to figure out the type of content and display it nicely for you:

```
$ git cat-file -p d670460b4b4aece5915caf5c68d12f560a9fe3e4
test content
```

Now, you can add content to Git and pull it back out again. You can also do this with content in files. For example, you can do some simple version control on a file. First, create a new file and save its contents in your database:

```
$ echo 'version 1' > test.txt
$ git hash-object -w test.txt
83baae61804e65cc73a7201a7252750c76066a30
```

Then, write some new content to the file, and save it again:

```
$ echo 'version 2' > test.txt
$ git hash-object -w test.txt
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
```

Your database contains the two new versions of the file as well as the first content you stored there:

```
$ find .git/objects -type f
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

Now you can revert the file back to the first version

```
$ git cat-file -p 83baae61804e65cc73a7201a7252750c76066a30 > test.txt
$ cat test.txt
version 1
```

or the second version:

```
$ git cat-file -p 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a > test.txt
$ cat test.txt
version 2
```

But remembering the SHA-1 key for each version of your file isn't practical; plus, you aren't storing the filename in your system – just the content. This ob-

ject type is called a blob. You can have Git tell you the object type of any object in Git, given its SHA-1 key, with `cat-file -t`:

```
$ git cat-file -t 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
blob
```

Tree Objects

The next type we'll look at is the tree, which solves the problem of storing the filename and also allows you to store a group of files together. Git stores content in a manner similar to a UNIX filesystem, but a bit simplified. All the content is stored as tree and blob objects, with trees corresponding to UNIX directory entries and blobs corresponding more or less to inodes or file contents. A single tree object contains one or more tree entries, each of which contains a SHA-1 pointer to a blob or subtree with its associated mode, type, and filename. For example, the most recent tree in a project may look something like this:

```
$ git cat-file -p master^{tree}
100644 blob a906cb2a4a904a152e80877d4088654daad0c859      README
100644 blob 8f94139338f9404f26296befa88755fc2598c289      Rakefile
040000 tree 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0      lib
```

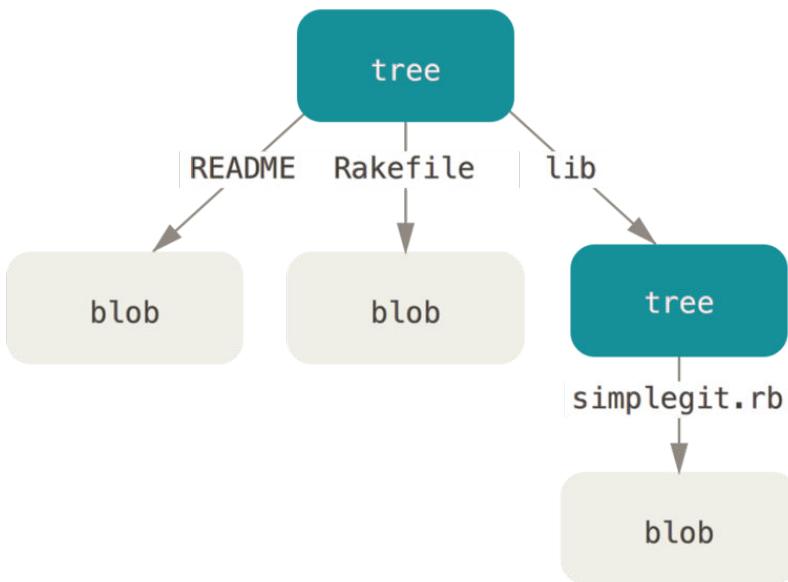
The `master^{tree}` syntax specifies the tree object that is pointed to by the last commit on your `master` branch. Notice that the `lib` subdirectory isn't a blob but a pointer to another tree:

```
$ git cat-file -p 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0
100644 blob 47c6340d6459e05787f644c2447d2595f5d3a54b      simplegit.rb
```

Conceptually, the data that Git is storing is something like this:

FIGURE 10-1

Simple version of the Git data model.



You can fairly easily create your own tree. Git normally creates a tree by taking the state of your staging area or index and writing a series of tree objects from it. So, to create a tree object, you first have to set up an index by staging some files. To create an index with a single entry – the first version of your test.txt file – you can use the plumbing command `update-index`. You use this command to artificially add the earlier version of the test.txt file to a new staging area. You must pass it the `--add` option because the file doesn't yet exist in your staging area (you don't even have a staging area set up yet) and `--cacheinfo` because the file you're adding isn't in your directory but is in your database. Then, you specify the mode, SHA-1, and filename:

```
$ git update-index --add --cacheinfo 100644 \
83baae61804e65cc73a7201a7252750c76066a30 test.txt
```

In this case, you're specifying a mode of `100644`, which means it's a normal file. Other options are `100755`, which means it's an executable file; and `120000`, which specifies a symbolic link. The mode is taken from normal UNIX modes but is much less flexible – these three modes are the only ones that are valid for files (blobs) in Git (although other modes are used for directories and submodules).

Now, you can use the `write-tree` command to write the staging area out to a tree object. No `-w` option is needed – calling `write-tree` automatically creates a tree object from the state of the index if that tree doesn't yet exist:

```
$ git write-tree
d8329fc1cc938780ffdd9f94e0d364e0ea74f579
$ git cat-file -p d8329fc1cc938780ffdd9f94e0d364e0ea74f579
100644 blob 83baae61804e65cc73a7201a7252750c76066a30      test.txt
```

You can also verify that this is a tree object:

```
$ git cat-file -t d8329fc1cc938780ffdd9f94e0d364e0ea74f579
tree
```

You'll now create a new tree with the second version of `test.txt` and a new file as well:

```
$ echo 'new file' > new.txt
$ git update-index test.txt
$ git update-index --add new.txt
```

Your staging area now has the new version of `test.txt` as well as the new file `new.txt`. Write out that tree (recording the state of the staging area or index to a tree object) and see what it looks like:

```
$ git write-tree
0155eb4229851634a0f03eb265b69f5a2d56f341
$ git cat-file -p 0155eb4229851634a0f03eb265b69f5a2d56f341
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a      test.txt
```

Notice that this tree has both file entries and also that the `test.txt` SHA-1 is the “version 2” SHA-1 from earlier (`1f7a7a`). Just for fun, you’ll add the first tree as a subdirectory into this one. You can read trees into your staging area by calling `read-tree`. In this case, you can read an existing tree into your staging area as a subtree by using the `--prefix` option to `read-tree`:

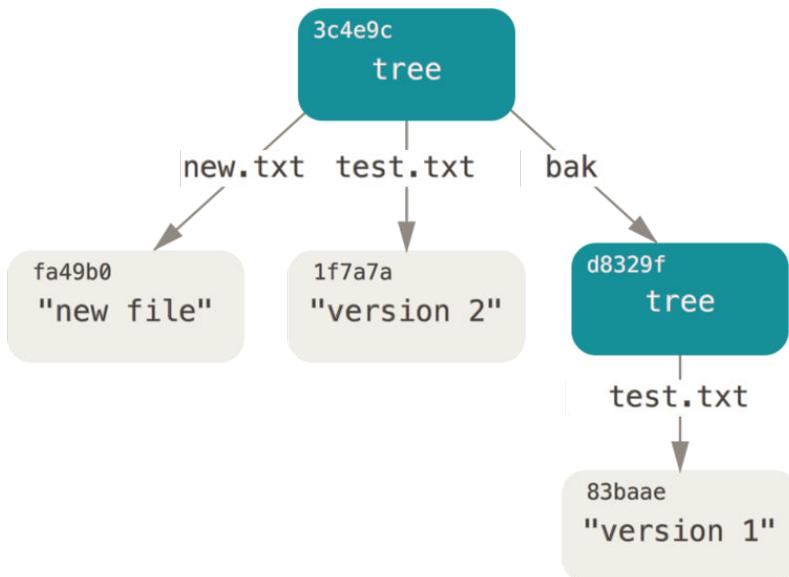
```
$ git read-tree --prefix=bak d8329fc1cc938780ffdd9f94e0d364e0ea74f579
$ git write-tree
3c4e9cd789d88d8d89c1073707c3585e41b0e614
```

```
$ git cat-file -p 3c4e9cd789d88d8d89c1073707c3585e41b0e614
040000 tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579      bak
100644 blob fa49b077972391ad58037050f2a75f74e3671e92    new.txt
100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a    test.txt
```

If you created a working directory from the new tree you just wrote, you would get the two files in the top level of the working directory and a subdirectory named `bak` that contained the first version of the `test.txt` file. You can think of the data that Git contains for these structures as being like this:

FIGURE 10-2

The content structure of your current Git data.



Commit Objects

You have three trees that specify the different snapshots of your project that you want to track, but the earlier problem remains: you must remember all three SHA-1 values in order to recall the snapshots. You also don't have any information about who saved the snapshots, when they were saved, or why they were saved. This is the basic information that the commit object stores for you.

To create a commit object, you call `commit-tree` and specify a single tree SHA-1 and which commit objects, if any, directly preceded it. Start with the first tree you wrote:

```
$ echo 'first commit' | git commit-tree d8329f
fdf4fc3344e67ab068f836878b6c4951e3b15f3d
```

Now you can look at your new commit object with `cat-file`:

```
$ git cat-file -p fdf4fc3
tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579
author Scott Chacon <schacon@gmail.com> 1243040974 -0700
committer Scott Chacon <schacon@gmail.com> 1243040974 -0700

first commit
```

The format for a commit object is simple: it specifies the top-level tree for the snapshot of the project at that point; the author/committer information (which uses your `user.name` and `user.email` configuration settings and a timestamp); a blank line, and then the commit message.

Next, you'll write the other two commit objects, each referencing the commit that came directly before it:

```
$ echo 'second commit' | git commit-tree 0155eb -p fdf4fc3
cac0cab538b970a37ea1e769cbbde608743bc96d
$ echo 'third commit' | git commit-tree 3c4e9c -p cac0cab
1a410efbd13591db07496601ebc7a059dd55cfe9
```

Each of the three commit objects points to one of the three snapshot trees you created. Oddly enough, you have a real Git history now that you can view with the `git log` command, if you run it on the last commit SHA-1:

```
$ git log --stat 1a410e
commit 1a410efbd13591db07496601ebc7a059dd55cfe9
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:15:24 2009 -0700

third commit

bak/test.txt | 1 +
1 file changed, 1 insertion(+)

commit cac0cab538b970a37ea1e769cbbde608743bc96d
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:14:29 2009 -0700
```

```

second commit

new.txt | 1 +
test.txt | 2 ++
2 files changed, 2 insertions(+), 1 deletion(-)

commit fdf4fc3344e67ab068f836878b6c4951e3b15f3d
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:09:34 2009 -0700

first commit

test.txt | 1 +
1 file changed, 1 insertion(+)

```

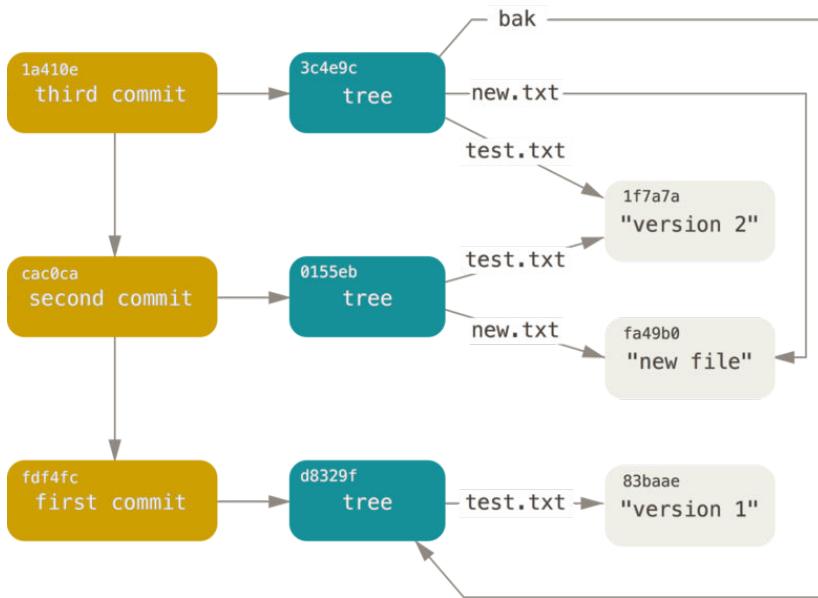
Amazing. You've just done the low-level operations to build up a Git history without using any of the front end commands. This is essentially what Git does when you run the `git add` and `git commit` commands – it stores blobs for the files that have changed, updates the index, writes out trees, and writes commit objects that reference the top-level trees and the commits that came immediately before them. These three main Git objects – the blob, the tree, and the commit – are initially stored as separate files in your `.git/objects` directory. Here are all the objects in the example directory now, commented with what they store:

```

$ find .git/objects -type f
.git/objects/01/55eb4229851634a0f03eb265b69f5a2d56f341 # tree 2
.git/objects/1a/410efbd13591db07496601ebc7a059dd55cf9 # commit 3
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a # test.txt v2
.git/objects/3c/4e9cd789d88d8d89c1073707c3585e41b0e614 # tree 3
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30 # test.txt v1
.git/objects/ca/c0cab538b970a37ea1e769cbbde608743bc96d # commit 2
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4 # 'test content'
.git/objects/d8/329fc1cc938780ffdd9f94e0d364e0ea74f579 # tree 1
.git/objects/fa/49b077972391ad58037050f2a75f74e3671e92 # new.txt
.git/objects/fd/f4fc3344e67ab068f836878b6c4951e3b15f3d # commit 1

```

If you follow all the internal pointers, you get an object graph something like this:

**FIGURE 10-3**

All the objects in your Git directory.

Object Storage

We mentioned earlier that a header is stored with the content. Let's take a minute to look at how Git stores its objects. You'll see how to store a blob object – in this case, the string “what is up, doc?” – interactively in the Ruby scripting language.

You can start up interactive Ruby mode with the `irb` command:

```
$ irb
>> content = "what is up, doc?"
=> "what is up, doc?"
```

Git constructs a header that starts with the type of the object, in this case a blob. Then, it adds a space followed by the size of the content and finally a null byte:

```
>> header = "blob #{content.length}\0"
=> "blob 16\0000"
```

Git concatenates the header and the original content and then calculates the SHA-1 checksum of that new content. You can calculate the SHA-1 value of a string in Ruby by including the SHA1 digest library with the `require` command and then calling `Digest::SHA1hexdigest()` with the string:

```
>> store = header + content
=> "blob 16\u0000what is up, doc?"
>> require 'digest/sha1'
=> true
>> sha1 = Digest::SHA1hexdigest(store)
=> "bd9dbf5aae1a3862dd1526723246b20206e5fc37"
```

Git compresses the new content with zlib, which you can do in Ruby with the `zlib` library. First, you need to require the library and then run `Zlib::Deflate.deflate()` on the content:

```
>> require 'zlib'
=> true
>> zlib_content = Zlib::Deflate.deflate(store)
=> "x\x9C\xCA\xC90R04c(\xFCH,Q\xC8,V(-\xD0QH\xC90\xB6\ax\x00_\x1C\ax\x9D"
```

Finally, you'll write your zlib-deflated content to an object on disk. You'll determine the path of the object you want to write out (the first two characters of the SHA-1 value being the subdirectory name, and the last 38 characters being the filename within that directory). In Ruby, you can use the `FileUtils.mkdir_p()` function to create the subdirectory if it doesn't exist. Then, open the file with `File.open()` and write out the previously zlib-compressed content to the file with a `write()` call on the resulting file handle:

```
>> path = '.git/objects/' + sha1[0,2] + '/' + sha1[2,38]
=> ".git/objects/bd/9dbf5aae1a3862dd1526723246b20206e5fc37"
>> require 'fileutils'
=> true
>> FileUtils.mkdir_p(File.dirname(path))
=> ".git/objects/bd"
>> File.open(path, 'w') { |f| f.write zlib_content }
=> 32
```

That's it – you've created a valid Git blob object. All Git objects are stored the same way, just with different types – instead of the string blob, the header will

begin with commit or tree. Also, although the blob content can be nearly anything, the commit and tree content are very specifically formatted.

Git References

You can run something like `git log 1a410e` to look through your whole history, but you still have to remember that `1a410e` is the last commit in order to walk that history to find all those objects. You need a file in which you can store the SHA-1 value under a simple name so you can use that pointer rather than the raw SHA-1 value.

In Git, these are called “references” or “refs;” you can find the files that contain the SHA-1 values in the `.git/refs` directory. In the current project, this directory contains no files, but it does contain a simple structure:

```
$ find .git/refs
.git/refs
.git/refs/heads
.git/refs/tags
$ find .git/refs -type f
```

To create a new reference that will help you remember where your latest commit is, you can technically do something as simple as this:

```
$ echo "1a410efbd13591db07496601ebc7a059dd55cfe9" > .git/refs/heads/master
```

Now, you can use the head reference you just created instead of the SHA-1 value in your Git commands:

```
$ git log --pretty=oneline master
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

You aren’t encouraged to directly edit the reference files. Git provides a safer command to do this if you want to update a reference called `update-ref`:

```
$ git update-ref refs/heads/master 1a410efbd13591db07496601ebc7a059dd55cfe9
```

That's basically what a branch in Git is: a simple pointer or reference to the head of a line of work. To create a branch back at the second commit, you can do this:

```
$ git update-ref refs/heads/test cac0ca
```

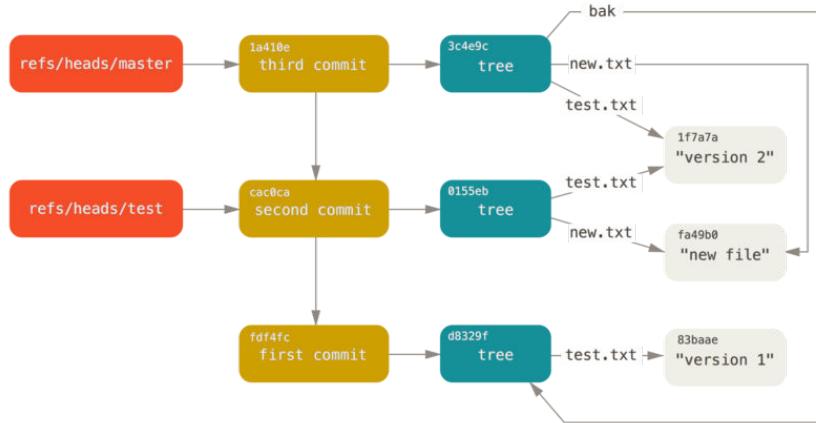
Your branch will contain only work from that commit down:

```
$ git log --pretty=oneline test
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Now, your Git database conceptually looks something like this:

FIGURE 10-4

Git directory objects with branch head references included.



When you run commands like `git branch (branchname)`, Git basically runs that `update-ref` command to add the SHA-1 of the last commit of the branch you're on into whatever new reference you want to create.

The HEAD

The question now is, when you run `git branch (branchname)`, how does Git know the SHA-1 of the last commit? The answer is the HEAD file.

The HEAD file is a symbolic reference to the branch you're currently on. By symbolic reference, we mean that unlike a normal reference, it doesn't general-

ly contain a SHA-1 value but rather a pointer to another reference. If you look at the file, you'll normally see something like this:

```
$ cat .git/HEAD  
ref: refs/heads/master
```

If you run `git checkout test`, Git updates the file to look like this:

```
$ cat .git/HEAD  
ref: refs/heads/test
```

When you run `git commit`, it creates the commit object, specifying the parent of that commit object to be whatever SHA-1 value the reference in HEAD points to.

You can also manually edit this file, but again a safer command exists to do so: `symbolic-ref`. You can read the value of your HEAD via this command:

```
$ git symbolic-ref HEAD  
refs/heads/master
```

You can also set the value of HEAD:

```
$ git symbolic-ref HEAD refs/heads/test  
$ cat .git/HEAD  
ref: refs/heads/test
```

You can't set a symbolic reference outside of the refs style:

```
$ git symbolic-ref HEAD test  
fatal: Refusing to point HEAD outside of refs/
```

Tags

We just finished discussing Git's three main object types, but there is a fourth. The tag object is very much like a commit object – it contains a tagger, a date, a message, and a pointer. The main difference is that a tag object generally points to a commit rather than a tree. It's like a branch reference, but it never moves – it always points to the same commit but gives it a friendlier name.

As discussed in [Chapter 2](#), there are two types of tags: annotated and lightweight. You can make a lightweight tag by running something like this:

```
$ git update-ref refs/tags/v1.0 cac0cab538b970a37ea1e769cbbde608743bc96d
```

That is all a lightweight tag is – a reference that never moves. An annotated tag is more complex, however. If you create an annotated tag, Git creates a tag object and then writes a reference to point to it rather than directly to the commit. You can see this by creating an annotated tag (-a specifies that it's an annotated tag):

```
$ git tag -a v1.1 1a410efbd13591db07496601ebc7a059dd55cfe9 -m 'test tag'
```

Here's the object SHA-1 value it created:

```
$ cat .git/refs/tags/v1.1
9585191f37f7b0fb9444f35a9bf50de191beadc2
```

Now, run the `cat-file` command on that SHA-1 value:

```
$ git cat-file -p 9585191f37f7b0fb9444f35a9bf50de191beadc2
object 1a410efbd13591db07496601ebc7a059dd55cfe9
type commit
tag v1.1
tagger Scott Chacon <schacon@gmail.com> Sat May 23 16:48:58 2009 -0700

test tag
```

Notice that the object entry points to the commit SHA-1 value that you tagged. Also notice that it doesn't need to point to a commit; you can tag any Git object. In the Git source code, for example, the maintainer has added their GPG public key as a blob object and then tagged it. You can view the public key by running this in a clone of the Git repository:

```
$ git cat-file blob junio-gpg-pub
```

The Linux kernel repository also has a non-commit-pointing tag object – the first tag created points to the initial tree of the import of the source code.

Remotes

The third type of reference that you'll see is a remote reference. If you add a remote and push to it, Git stores the value you last pushed to that remote for each branch in the `refs/remotes` directory. For instance, you can add a remote called `origin` and push your `master` branch to it:

```
$ git remote add origin git@github.com:schacon/simplegit-progit.git
$ git push origin master
Counting objects: 11, done.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (7/7), 716 bytes, done.
Total 7 (delta 2), reused 4 (delta 1)
To git@github.com:schacon/simplegit-progit.git
  a11bef0..ca82a6d  master -> master
```

Then, you can see what the `master` branch on the `origin` remote was the last time you communicated with the server, by checking the `refs/remotes/origin/master` file:

```
$ cat .git/refs/remotes/origin/master
ca82a6dff817ec66f44342007202690a93763949
```

Remote references differ from branches (`refs/heads` references) mainly in that they're considered read-only. You can `git checkout` to one, but Git won't point `HEAD` at one, so you'll never update it with a `commit` command. Git manages them as bookmarks to the last known state of where those branches were on those servers.

Packfiles

Let's go back to the objects database for your test Git repository. At this point, you have 11 objects – 4 blobs, 3 trees, 3 commits, and 1 tag:

```
$ find .git/objects -type f
.git/objects/01/55eb4229851634a0f03eb265b69f5a2d56f341 # tree 2
.git/objects/1a/410efbd13591db07496601ebc7a059dd55cfe9 # commit 3
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a # test.txt v2
.git/objects/3c/4e9cd789d88d889c1073707c3585e41b0e614 # tree 3
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30 # test.txt v1
.git/objects/95/85191f37f7b0fb9444f35a9bf50de191beadc2 # tag
```

```
.git/objects/ca/c0cab538b970a37ea1e769cbbde608743bc96d # commit 2
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4 # 'test content'
.git/objects/d8/329fc1cc938780ffdd9f94e0d364e0ea74f579 # tree 1
.git/objects/fa/49b077972391ad58037050f2a75f74e3671e92 # new.txt
.git/objects/fd/f4fc3344e67ab068f836878b6c4951e3b15f3d # commit 1
```

Git compresses the contents of these files with zlib, and you’re not storing much, so all these files collectively take up only 925 bytes. You’ll add some larger content to the repository to demonstrate an interesting feature of Git. To demonstrate, we’ll add the `repo.rb` file from the Grit library – this is about a 22K source code file:

```
$ curl https://raw.githubusercontent.com/mojombo/grit/master/lib/grit/repo.rb > repo.rb
$ git add repo.rb
$ git commit -m 'added repo.rb'
[master 484a592] added repo.rb
 3 files changed, 709 insertions(+), 2 deletions(-)
 delete mode 100644 bak/test.txt
 create mode 100644 repo.rb
 rewrite test.txt (100%)
```

If you look at the resulting tree, you can see the SHA-1 value your `repo.rb` file got for the blob object:

```
$ git cat-file -p master^{tree}
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob 033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5      repo.rb
100644 blob e3f094f522629ae358806b17daf78246c27c007b      test.txt
```

You can then use `git cat-file` to see how big that object is:

```
$ git cat-file -s 033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5
22044
```

Now, modify that file a little, and see what happens:

```
$ echo '# testing' >> repo.rb
$ git commit -am 'modified repo a bit'
[master 2431da6] modified repo.rb a bit
 1 file changed, 1 insertion(+)
```

Check the tree created by that commit, and you see something interesting:

```
$ git cat-file -p master^{tree}
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob b042a60ef7dff760008df33cee372b945b6e884e      repo.rb
100644 blob e3f094f522629ae358806b17daf78246c27c007b      test.txt
```

The blob is now a different blob, which means that although you added only a single line to the end of a 400-line file, Git stored that new content as a completely new object:

```
$ git cat-file -s b042a60ef7dff760008df33cee372b945b6e884e
22054
```

You have two nearly identical 22K objects on your disk. Wouldn't it be nice if Git could store one of them in full but then the second object only as the delta between it and the first?

It turns out that it can. The initial format in which Git saves objects on disk is called a "loose" object format. However, occasionally Git packs up several of these objects into a single binary file called a "packfile" in order to save space and be more efficient. Git does this if you have too many loose objects around, if you run the `git gc` command manually, or if you push to a remote server. To see what happens, you can manually ask Git to pack up the objects by calling the `git gc` command:

```
$ git gc
Counting objects: 18, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (14/14), done.
Writing objects: 100% (18/18), done.
Total 18 (delta 3), reused 0 (delta 0)
```

If you look in your `objects` directory, you'll find that most of your objects are gone, and a new pair of files has appeared:

```
$ find .git/objects -type f
.git/objects/bd/9dbf5aae1a3862dd1526723246b20206e5fc37
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
.git/objects/info/packs
.git/objects/pack/pack-978e03944f5c581011e6998cd0e9e30000905586.idx
.git/objects/pack/pack-978e03944f5c581011e6998cd0e9e30000905586.pack
```

The objects that remain are the blobs that aren't pointed to by any commit – in this case, the “what is up, doc?” example and the “test content” example blobs you created earlier. Because you never added them to any commits, they're considered dangling and aren't packed up in your new packfile.

The other files are your new packfile and an index. The packfile is a single file containing the contents of all the objects that were removed from your filesystem. The index is a file that contains offsets into that packfile so you can quickly seek to a specific object. What is cool is that although the objects on disk before you ran the gc were collectively about 22K in size, the new packfile is only 7K. You've cut your disk usage by $\frac{1}{3}$ by packing your objects.

How does Git do this? When Git packs objects, it looks for files that are named and sized similarly, and stores just the deltas from one version of the file to the next. You can look into the packfile and see what Git did to save space. The `git verify-pack` plumbing command allows you to see what was packed up:

```
$ git verify-pack -v .git/objects/pack/pack-978e03944f5c581011e6998cd0e9e300009055
2431da676938450a4d72e260db3bf7b0f587bbc1 commit 223 155 12
69bcdaff5328278ab1c0812ce0e07fa7d26a96d7 commit 214 152 167
80d02664cb23ed55b226516648c7ad5d0a3deb90 commit 214 145 319
43168a18b7613d1281e5560855a83eb8fde3d687 commit 213 146 464
092917823486a802e94d727c820a9024e14a1fc2 commit 214 146 610
702470739ce72005e2edff522fde85d52a65df9b commit 165 118 756
d368d0ac0678cbc6cce505be58126d3526706e54 tag 130 122 874
fe879577cb8cffcdf25441725141e310dd7d239b tree 136 136 996
d8329fc1cc938780ffdd9f94e0d364e0ea74f579 tree 36 46 1132
deef2e1b793907545e50a2ea2ddb5ba6c58c4506 tree 136 136 1178
d982c7cb2c2a972ee391a85da481fc1f9127a01d tree 6 17 1314 1 \
    deef2e1b793907545e50a2ea2ddb5ba6c58c4506
3c4e9cd789d88d8d89c1073707c3585e41b0e614 tree 8 19 1331 1 \
        deef2e1b793907545e50a2ea2ddb5ba6c58c4506
0155eb4229851634a0f03eb265b69f5a2d56f341 tree 71 76 1350
83baae61804e65cc73a7201a7252750c76066a30 blob 10 19 1426
fa49b077972391ad58037050f2a75f74e3671e92 blob 9 18 1445
b042a60ef7dff760008df33cee372b945b6e884e blob 22054 5799 1463
033b4468fa6b2a9547a70d88d1bbe8bf3f9ed05 blob 9 20 7262 1 \
    b042a60ef7dff760008df33cee372b945b6e884e
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a blob 10 19 7282
non delta: 15 objects
chain length = 1: 3 objects
.git/objects/pack/pack-978e03944f5c581011e6998cd0e9e30000905586.pack: ok
```

Here, the `033b4` blob, which if you remember was the first version of your `repo.rb` file, is referencing the `b042a` blob, which was the second version of the file. The third column in the output is the size of the object in the pack, so you

can see that b042a takes up 22K of the file, but that 033b4 only takes up 9 bytes. What is also interesting is that the second version of the file is the one that is stored intact, whereas the original version is stored as a delta – this is because you’re most likely to need faster access to the most recent version of the file.

The really nice thing about this is that it can be repacked at any time. Git will occasionally repack your database automatically, always trying to save more space, but you can also manually repack at any time by running `git gc` by hand.

The Refspec

Throughout this book, we’ve used simple mappings from remote branches to local references, but they can be more complex. Suppose you add a remote like this:

```
$ git remote add origin https://github.com/schacon/simplegit-progit
```

It adds a section to your `.git/config` file, specifying the name of the remote (`origin`), the URL of the remote repository, and the refspec for fetching:

```
[remote "origin"]
url = https://github.com/schacon/simplegit-progit
fetch = +refs/heads/*:refs/remotes/origin/*
```

The format of the refspec is an optional `+`, followed by `<src>:<dst>`, where `<src>` is the pattern for references on the remote side and `<dst>` is where those references will be written locally. The `+` tells Git to update the reference even if it isn’t a fast-forward.

In the default case that is automatically written by a `git remote add` command, Git fetches all the references under `refs/heads/` on the server and writes them to `refs/remotes/origin/` locally. So, if there is a `master` branch on the server, you can access the log of that branch locally via

```
$ git log origin/master
$ git log remotes/origin/master
$ git log refs/remotes/origin/master
```

They’re all equivalent, because Git expands each of them to `refs/remotes/origin/master`.

If you want Git instead to pull down only the `master` branch each time, and not every other branch on the remote server, you can change the fetch line to

```
fetch = +refs/heads/master:refs/remotes/origin/master
```

This is just the default refspec for `git fetch` for that remote. If you want to do something one time, you can specify the refspec on the command line, too. To pull the `master` branch on the remote down to `origin/mymaster` locally, you can run

```
$ git fetch origin master:refs/remotes/origin/mymaster
```

You can also specify multiple refspecs. On the command line, you can pull down several branches like so:

```
$ git fetch origin master:refs/remotes/origin/mymaster \
    topic:refs/remotes/origin/topic
From git@github.com:schacon/simplegit
! [rejected]      master      -> origin/mymaster (non fast forward)
* [new branch]    topic       -> origin/topic
```

In this case, the `master` branch pull was rejected because it wasn't a fast-forward reference. You can override that by specifying the `+` in front of the refspec.

You can also specify multiple refspecs for fetching in your configuration file. If you want to always fetch the `master` and `experiment` branches, add two lines:

```
[remote "origin"]
  url = https://github.com/schacon/simplegit-progit
  fetch = +refs/heads/master:refs/remotes/origin/master
  fetch = +refs/heads/experiment:refs/remotes/origin/experiment
```

You can't use partial globs in the pattern, so this would be invalid:

```
fetch = +refs/heads/qa*:refs/remotes/origin/qa*
```

However, you can use namespaces (or directories) to accomplish something like that. If you have a QA team that pushes a series of branches, and you want to get the `master` branch and any of the QA team's branches but nothing else, you can use a config section like this:

```
[remote "origin"]
  url = https://github.com/schacon/simplegit-progit
  fetch = +refs/heads/master:refs/remotes/origin/master
  fetch = +refs/heads/qa/*:refs/remotes/origin/qa/*
```

If you have a complex workflow process that has a QA team pushing branches, developers pushing branches, and integration teams pushing and collaborating on remote branches, you can namespace them easily this way.

Pushing Refspecs

It's nice that you can fetch namespaced references that way, but how does the QA team get their branches into a qa/ namespace in the first place? You accomplish that by using refspecs to push.

If the QA team wants to push their master branch to qa/master on the remote server, they can run

```
$ git push origin master:refs/heads/qa/master
```

If they want Git to do that automatically each time they run `git push origin`, they can add a push value to their config file:

```
[remote "origin"]
  url = https://github.com/schacon/simplegit-progit
  fetch = +refs/heads/*:refs/remotes/origin/*
  push = refs/heads/master:refs/heads/qa/master
```

Again, this will cause a `git push origin` to push the local master branch to the remote qa/master branch by default.

Deleting References

You can also use the refspec to delete references from the remote server by running something like this:

```
$ git push origin :topic
```

Because the refspec is `<src>:<dst>`, by leaving off the `<src>` part, this basically says to make the topic branch on the remote nothing, which deletes it.

Transfer Protocols

Git can transfer data between two repositories in two major ways: the “dumb” protocol and the “smart” protocol. This section will quickly cover how these two main protocols operate.

The Dumb Protocol

If you’re setting up a repository to be served read-only over HTTP, the dumb protocol is likely what will be used. This protocol is called “dumb” because it requires no Git-specific code on the server side during the transport process; the fetch process is a series of HTTP GET requests, where the client can assume the layout of the Git repository on the server.

The dumb protocol is fairly rarely used these days. It’s difficult to secure or make private, so most Git hosts (both cloud-based and on-premises) will refuse to use it. It’s generally advised to use the smart protocol, which we describe a bit further on.

Let’s follow the `http-fetch` process for the `simplegit` library:

```
$ git clone http://server/simplegit-progit.git
```

The first thing this command does is pull down the `info/refs` file. This file is written by the `update-server-info` command, which is why you need to enable that as a `post-receive` hook in order for the HTTP transport to work properly:

```
=> GET info/refs  
ca82a6dff817ec66f44342007202690a93763949 refs/heads/master
```

Now you have a list of the remote references and SHA-1s. Next, you look for what the `HEAD` reference is so you know what to check out when you’re finished:

```
=> GET HEAD  
ref: refs/heads/master
```

You need to check out the `master` branch when you’ve completed the process. At this point, you’re ready to start the walking process. Because your start-

ing point is the `ca82a6` commit object you saw in the `info/refs` file, you start by fetching that:

```
=> GET objects/ca/82a6dff817ec66f44342007202690a93763949
(179 bytes of binary data)
```

You get an object back – that object is in loose format on the server, and you fetched it over a static HTTP GET request. You can zlib-uncompress it, strip off the header, and look at the commit content:

```
$ git cat-file -p ca82a6dff817ec66f44342007202690a93763949
tree cfda3bf379e4f8dba8717dee55aab78aef7f4daf
parent 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
author Scott Chacon <schacon@gmail.com> 1205815931 -0700
committer Scott Chacon <schacon@gmail.com> 1240030591 -0700

changed the version number
```

Next, you have two more objects to retrieve – `cfda3b`, which is the tree of content that the commit we just retrieved points to; and `085bb3`, which is the parent commit:

```
=> GET objects/08/5bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
(179 bytes of data)
```

That gives you your next commit object. Grab the tree object:

```
=> GET objects/cf/da3bf379e4f8dba8717dee55aab78aef7f4daf
(404 - Not Found)
```

Oops – it looks like that tree object isn't in loose format on the server, so you get a 404 response back. There are a couple of reasons for this – the object could be in an alternate repository, or it could be in a packfile in this repository. Git checks for any listed alternates first:

```
=> GET objects/info/http-alternates
(empty file)
```

If this comes back with a list of alternate URLs, Git checks for loose files and packfiles there – this is a nice mechanism for projects that are forks of one another to share objects on disk. However, because no alternates are listed in this case, your object must be in a packfile. To see what packfiles are available on this server, you need to get the `objects/info/packs` file, which contains a listing of them (also generated by `update-server-info`):

```
=> GET objects/info/packs
P pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack
```

There is only one packfile on the server, so your object is obviously in there, but you'll check the index file to make sure. This is also useful if you have multiple packfiles on the server, so you can see which packfile contains the object you need:

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.idx
(4k of binary data)
```

Now that you have the packfile index, you can see if your object is in it – because the index lists the SHA-1s of the objects contained in the packfile and the offsets to those objects. Your object is there, so go ahead and get the whole packfile:

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack
(13k of binary data)
```

You have your tree object, so you continue walking your commits. They're all also within the packfile you just downloaded, so you don't have to do any more requests to your server. Git checks out a working copy of the `master` branch that was pointed to by the `HEAD` reference you downloaded at the beginning.

The Smart Protocol

The dumb protocol is simple but a bit inefficient, and it can't handle writing of data from the client to the server. The smart protocol is a more common method of transferring data, but it requires a process on the remote end that is intelligent about Git – it can read local data, figure out what the client has and needs, and generate a custom packfile for it. There are two sets of processes for transferring data: a pair for uploading data and a pair for downloading data.

UPLOADING DATA

To upload data to a remote process, Git uses the `send-pack` and `receive-pack` processes. The `send-pack` process runs on the client and connects to a `receive-pack` process on the remote side.

SSH

For example, say you run `git push origin master` in your project, and `origin` is defined as a URL that uses the SSH protocol. Git fires up the `send-pack` process, which initiates a connection over SSH to your server. It tries to

run a command on the remote server via an SSH call that looks something like this:

```
$ ssh -x git@server "git-receive-pack 'simplegit-progit.git'"  
005bca82a6dff817ec66f4437202690a93763949 refs/heads/master report-status \  
    delete-refs side-band-64k quiet ofs-delta \  
    agent=git/2:2.1.1+github-607-gfba4028 delete-refs  
003e085bb3bcb608e1e84b2432f8ecbe6306e7e7 refs/heads/topic  
0000
```

The `git-receive-pack` command immediately responds with one line for each reference it currently has – in this case, just the `master` branch and its SHA-1. The first line also has a list of the server's capabilities (here, `report-status`, `delete-refs`, and some others, including the client identifier).

Each line starts with a 4-character hex value specifying how long the rest of the line is. Your first line starts with 005b, which is hexadecimal for 91, meaning that 91 bytes remain on that line. The next line starts with 003e, which is 62, so you read the remaining 62 bytes. The next line is 0000, meaning the server is done with its references listing.

Now that it knows the server's state, your send-pack process determines what commits it has that the server doesn't. For each reference that this push will update, the send-pack process tells the receive-pack process that information. For instance, if you're updating the master branch and adding an experiment branch, the send-pack response may look something like this:

Git sends a line for each reference you're updating with the line's length, the old SHA-1, the new SHA-1, and the reference that is being updated. The first line also has the client's capabilities. The SHA-1 value of all '0's means that nothing was there before – because you're adding the experiment reference. If you were deleting a reference, you would see the opposite: all '0's on the right side.

Next, the client sends a packfile of all the objects the server doesn't have yet. Finally, the server responds with a success (or failure) indication:

000Aunpack ok

HTTP(S)

This process is mostly the same over HTTP, though the handshaking is a bit different. The connection is initiated with this request:

```
=> GET http://server/simplegit-progit.git/info/refs?service=git-receive-pack
001f# service=git-receive-pack
000000ab6c5f0e45abd7832bf23074a333f739977c9e8188 refs/heads/master \
report-status delete-refs side-band-64k quiet ofs-delta \
agent=git/2:2.1.1~vmg-bitmaps-bugaloo-608-g116744e
0000
```

That's the end of the first client-server exchange. The client then makes another request, this time a POST, with the data that `git-upload-pack` provides.

```
=> POST http://server/simplegit-progit.git/git-receive-pack
```

The POST request includes the `send-pack` output and the packfile as its payload. The server then indicates success or failure with its HTTP response.

DOWNLOADING DATA

When you download data, the `fetch-pack` and `upload-pack` processes are involved. The client initiates a `fetch-pack` process that connects to an `upload-pack` process on the remote side to negotiate what data will be transferred down.

SSH

If you're doing the fetch over SSH, `fetch-pack` instead runs something like this:

```
$ ssh -x git@server "git-upload-pack 'simplegit-progit.git'"
```

After `fetch-pack` connects, `upload-pack` sends back something like this:

```
00dfca82a6dff817ec66f44342007202690a93763949 HEADmulti_ack thin-pack \
side-band side-band-64k ofs-delta shallow no-progress include-tag \
multi_ack_detailed symref=HEAD:refs/heads/master \
agent=git/2:2.1.1+github-607-gfba4028
003fca82a6dff817ec66f44342007202690a93763949 refs/heads/master
0000
```

This is very similar to what `receive-pack` responds with, but the capabilities are different. In addition, it sends back what `HEAD` points to (`sym-`

`ref=HEAD:refs/heads/master)` so the client knows what to check out if this is a clone.

At this point, the `fetch-pack` process looks at what objects it has and responds with the objects that it needs by sending “want” and then the SHA-1 it wants. It sends all the objects it already has with “have” and then the SHA-1. At the end of this list, it writes “done” to initiate the `upload-pack` process to begin sending the packfile of the data it needs:

```
0054want ca82a6dff817ec66f44342007202690a93763949 ofs-delta
0032have 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
0000
0009done
```

HTTP(S)

The handshake for a fetch operation takes two HTTP requests. The first is a GET to the same endpoint used in the dumb protocol:

```
=> GET $GIT_URL/info/refs?service=git-upload-pack
001e# service=git-upload-pack
000000e7ca82a6dff817ec66f44342007202690a93763949 HEADmulti_ack thin-pack \
    side-band side-band-64k ofs-delta shallow no-progress include-tag \
    multi_ack_detailed no-done symref=HEAD:refs/heads/master \
    agent=git/2:2.1.1+github-607-gfba4028
003fcfa82a6dff817ec66f44342007202690a93763949 refs/heads/master
0000
```

This is very similar to invoking `git-upload-pack` over an SSH connection, but the second exchange is performed as a separate request:

```
=> POST $GIT_URL/git-upload-pack HTTP/1.0
0032want 0a53e9ddeaddad63ad106860237bbf53411d11a7
0032have 441b40d833fd9a93eb2908e5274224faf0ee993
0000
```

Again, this is the same format as above. The response to this request indicates success or failure, and includes the packfile.

Protocols Summary

This section contains a very basic overview of the transfer protocols. The protocol includes many other features, such as `multi_ack` or `side-band` capabilities, but covering them is outside the scope of this book. We've tried to give you a sense of the general back-and-forth between client and server; if you need

more knowledge than this, you'll probably want to take a look at the Git source code.

Maintenance and Data Recovery

Occasionally, you may have to do some cleanup – make a repository more compact, clean up an imported repository, or recover lost work. This section will cover some of these scenarios.

Maintenance

Occasionally, Git automatically runs a command called “auto gc”. Most of the time, this command does nothing. However, if there are too many loose objects (objects not in a packfile) or too many packfiles, Git launches a full-fledged git gc command. The “gc” stands for garbage collect, and the command does a number of things: it gathers up all the loose objects and places them in packfiles, it consolidates packfiles into one big packfile, and it removes objects that aren't reachable from any commit and are a few months old.

You can run auto gc manually as follows:

```
$ git gc --auto
```

Again, this generally does nothing. You must have around 7,000 loose objects or more than 50 packfiles for Git to fire up a real gc command. You can modify these limits with the gc.auto and gc.autopacklimit config settings, respectively.

The other thing gc will do is pack up your references into a single file. Suppose your repository contains the following branches and tags:

```
$ find .git/refs -type f
.git/refs/heads/experiment
.git/refs/heads/master
.git/refs/tags/v1.0
.git/refs/tags/v1.1
```

If you run git gc, you'll no longer have these files in the refs directory. Git will move them for the sake of efficiency into a file named .git/packed-refs that looks like this:

```
$ cat .git/packed-refs
# pack-refs with: peeled fully-peeled
cac0cab538b970a37ea1e769cbde608743bc96d refs/heads/experiment
ab1afef80fac8e34258ff41fc1b867c702daa24b refs/heads/master
cac0cab538b970a37ea1e769cbde608743bc96d refs/tags/v1.0
9585191f37f7b0fb9444f35a9bf50de191beadc2 refs/tags/v1.1
^1a410efbd13591db07496601ebc7a059dd55cfe9
```

If you update a reference, Git doesn't edit this file but instead writes a new file to `refs/heads`. To get the appropriate SHA-1 for a given reference, Git checks for that reference in the `refs` directory and then checks the `packed-refs` file as a fallback. However, if you can't find a reference in the `refs` directory, it's probably in your `packed-refs` file.

Notice the last line of the file, which begins with a `^`. This means the tag directly above is an annotated tag and that line is the commit that the annotated tag points to.

Data Recovery

At some point in your Git journey, you may accidentally lose a commit. Generally, this happens because you force-delete a branch that had work on it, and it turns out you wanted the branch after all; or you hard-reset a branch, thus abandoning commits that you wanted something from. Assuming this happens, how can you get your commits back?

Here's an example that hard-resets the `master` branch in your test repository to an older commit and then recovers the lost commits. First, let's review where your repository is at this point:

```
$ git log --pretty=oneline
ab1afef80fac8e34258ff41fc1b867c702daa24b modified repo a bit
484a59275031909e19aadb7c92262719cfcdf19a added repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Now, move the `master` branch back to the middle commit:

```
$ git reset --hard 1a410efbd13591db07496601ebc7a059dd55cfe9
HEAD is now at 1a410ef third commit
$ git log --pretty=oneline
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
```

```
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

You've effectively lost the top two commits – you have no branch from which those commits are reachable. You need to find the latest commit SHA-1 and then add a branch that points to it. The trick is finding that latest commit SHA-1 – it's not like you've memorized it, right?

Often, the quickest way is to use a tool called `git reflog`. As you're working, Git silently records what your HEAD is every time you change it. Each time you commit or change branches, the reflog is updated. The reflog is also updated by the `git update-ref` command, which is another reason to use it instead of just writing the SHA-1 value to your ref files, as we covered in “[Git References](#)”. You can see where you've been at any time by running `git reflog`:

```
$ git reflog
1a410ef HEAD@{0}: reset: moving to 1a410ef
ab1afef HEAD@{1}: commit: modified repo.rb a bit
484a592 HEAD@{2}: commit: added repo.rb
```

Here we can see the two commits that we have had checked out, however there is not much information here. To see the same information in a much more useful way, we can run `git log -g`, which will give you a normal log output for your reflog.

```
$ git log -g
commit 1a410efbd13591db07496601ebc7a059dd55cfe9
Reflog: HEAD@{0} (Scott Chacon <schacon@gmail.com>)
Reflog message: updating HEAD
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:22:37 2009 -0700

            third commit

commit ab1afef80fac8e34258ff41fc1b867c702daa24b
Reflog: HEAD@{1} (Scott Chacon <schacon@gmail.com>)
Reflog message: updating HEAD
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:15:24 2009 -0700

            modified repo.rb a bit
```

It looks like the bottom commit is the one you lost, so you can recover it by creating a new branch at that commit. For example, you can start a branch named `recover-branch` at that commit (`ab1afef`):

```
$ git branch recover-branch ab1afef
$ git log --pretty=oneline recover-branch
ab1afef80fac8e34258ff41fc1b867c702daa24b modified repo a bit
484a59275031909e19aadb7c92262719cfcd19a added repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbdde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Cool – now you have a branch named `recover-branch` that is where your `master` branch used to be, making the first two commits reachable again. Next, suppose your loss was for some reason not in the reflog – you can simulate that by removing `recover-branch` and deleting the reflog. Now the first two commits aren't reachable by anything:

```
$ git branch -D recover-branch
$ rm -Rf .git/logs/
```

Because the reflog data is kept in the `.git/logs/` directory, you effectively have no reflog. How can you recover that commit at this point? One way is to use the `git fsck` utility, which checks your database for integrity. If you run it with the `--full` option, it shows you all objects that aren't pointed to by another object:

```
$ git fsck --full
Checking object directories: 100% (256/256), done.
Checking objects: 100% (18/18), done.
dangling blob d670460b4b4aece5915caf5c68d12f560a9fe3e4
dangling commit ab1afef80fac8e34258ff41fc1b867c702daa24b
dangling tree aea790b9a58f6cf6f2804eeac9f0abbe9631e4c9
dangling blob 7108f7ecb345ee9d0084193f147cdad4d2998293
```

In this case, you can see your missing commit after the string “dangling commit”. You can recover it the same way, by adding a branch that points to that SHA-1.

Removing Objects

There are a lot of great things about Git, but one feature that can cause issues is the fact that a `git clone` downloads the entire history of the project, including every version of every file. This is fine if the whole thing is source code, because Git is highly optimized to compress that data efficiently. However, if someone at any point in the history of your project added a single huge file, every clone for all time will be forced to download that large file, even if it was removed from the project in the very next commit. Because it's reachable from the history, it will always be there.

This can be a huge problem when you're converting Subversion or Perforce repositories into Git. Because you don't download the whole history in those systems, this type of addition carries few consequences. If you did an import from another system or otherwise find that your repository is much larger than it should be, here is how you can find and remove large objects.

Be warned: this technique is destructive to your commit history. It re-writes every commit object since the earliest tree you have to modify to remove a large file reference. If you do this immediately after an import, before anyone has started to base work on the commit, you're fine – otherwise, you have to notify all contributors that they must rebase their work onto your new commits.

To demonstrate, you'll add a large file into your test repository, remove it in the next commit, find it, and remove it permanently from the repository. First, add a large object to your history:

```
$ curl https://www.kernel.org/pub/software/scm/git/git-2.1.0.tar.gz > git.tgz
$ git add git.tgz
$ git commit -m 'add git tarball'
[master 7b30847] add git tarball
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 git.tgz
```

Oops – you didn't want to add a huge tarball to your project. Better get rid of it:

```
$ git rm git.tgz
rm 'git.tgz'
$ git commit -m 'oops - removed large tarball'
[master dadf725] oops - removed large tarball
 1 file changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 git.tgz
```

Now, gc your database and see how much space you're using:

```
$ git gc
Counting objects: 17, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (13/13), done.
Writing objects: 100% (17/17), done.
Total 17 (delta 1), reused 10 (delta 0)
```

You can run the `count-objects` command to quickly see how much space you're using:

```
$ git count-objects -v
count: 7
size: 32
in-pack: 17
packs: 1
size-pack: 4868
prune-packable: 0
garbage: 0
size-garbage: 0
```

The `size-pack` entry is the size of your packfiles in kilobytes, so you're using almost 5MB. Before the last commit, you were using closer to 2K – clearly, removing the file from the previous commit didn't remove it from your history. Every time anyone clones this repository, they will have to clone all 5MB just to get this tiny project, because you accidentally added a big file. Let's get rid of it.

First you have to find it. In this case, you already know what file it is. But suppose you didn't; how would you identify what file or files were taking up so much space? If you run `git gc`, all the objects are in a packfile; you can identify the big objects by running another plumbing command called `git verify-pack` and sorting on the third field in the output, which is file size. You can also pipe it through the `tail` command because you're only interested in the last few largest files:

```
$ git verify-pack -v .git/objects/pack/pack-29..69.idx \
| sort -k 3 -n \
| tail -3
dadf7258d699da2c8d89b09ef6670edb7d5f91b4 commit 229 159 12
033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5 blob 22044 5792 4977696
82c99a3e86bb1267b236a4b6eff7868d97489af1 blob 4975916 4976258 1438
```

The big object is at the bottom: 5MB. To find out what file it is, you'll use the `rev-list` command, which you used briefly in “**Obligando a utilizar un for-**

mato específico en el mensaje de commit”. If you pass `--objects` to `rev-list`, it lists all the commit SHA-1s and also the blob SHA-1s with the file paths associated with them. You can use this to find your blob’s name:

```
$ git rev-list --objects --all | grep 82c99a3
82c99a3e86bb1267b236a4b6eff7868d97489af1 git.tgz
```

Now, you need to remove this file from all trees in your past. You can easily see what commits modified this file:

```
$ git log --oneline --branches -- git.tgz
dadf725 oops - removed large tarball
7b30847 add git tarball
```

You must rewrite all the commits downstream from `7b30847` to fully remove this file from your Git history. To do so, you use `filter-branch`, which you used in “[Rewriting History](#)”:

```
$ git filter-branch --index-filter \
'git rm --cached --ignore-unmatch git.tgz' -- 7b30847^..
Rewrite 7b30847d080183a1ab7d18fb202473b3096e9f34 (1/2)rm 'git.tgz'
Rewrite dadf7258d699da2c8d89b09ef6670edb7d5f91b4 (2/2)
Ref 'refs/heads/master' was rewritten
```

The `--index-filter` option is similar to the `--tree-filter` option used in “[Rewriting History](#)”, except that instead of passing a command that modifies files checked out on disk, you’re modifying your staging area or index each time.

Rather than remove a specific file with something like `rm file`, you have to remove it with `git rm --cached` – you must remove it from the index, not from disk. The reason to do it this way is speed – because Git doesn’t have to check out each revision to disk before running your filter, the process can be much, much faster. You can accomplish the same task with `--tree-filter` if you want. The `--ignore-unmatch` option to `git rm` tells it not to error out if the pattern you’re trying to remove isn’t there. Finally, you ask `filter-branch` to rewrite your history only from the `7b30847` commit up, because you know that is where this problem started. Otherwise, it will start from the beginning and will unnecessarily take longer.

Your history no longer contains a reference to that file. However, your reflog and a new set of refs that Git added when you did the `filter-branch` un-

der .git/refs/original still do, so you have to remove them and then repack the database. You need to get rid of anything that has a pointer to those old commits before you repack:

```
$ rm -Rf .git/refs/original  
$ rm -Rf .git/logs/  
$ git gc  
Counting objects: 15, done.  
Delta compression using up to 8 threads.  
Compressing objects: 100% (11/11), done.  
Writing objects: 100% (15/15), done.  
Total 15 (delta 1), reused 12 (delta 0)
```

Let's see how much space you saved.

```
$ git count-objects -v  
count: 11  
size: 4904  
in-pack: 15  
packs: 1  
size-pack: 8  
prune-packable: 0  
garbage: 0  
size-garbage: 0
```

The packed repository size is down to 8K, which is much better than 5MB. You can see from the size value that the big object is still in your loose objects, so it's not gone; but it won't be transferred on a push or subsequent clone, which is what is important. If you really wanted to, you could remove the object completely by running `git prune` with the `--expire` option:

```
$ git prune --expire now  
$ git count-objects -v  
count: 0  
size: 0  
in-pack: 15  
packs: 1  
size-pack: 8  
prune-packable: 0  
garbage: 0  
size-garbage: 0
```

Environment Variables

Git always runs inside a bash shell, and uses a number of shell environment variables to determine how it behaves. Occasionally, it comes in handy to know what these are, and how they can be used to make Git behave the way you want it to. This isn't an exhaustive list of all the environment variables Git pays attention to, but we'll cover the most useful.

Global Behavior

Some of Git's general behavior as a computer program depends on environment variables.

GIT_EXEC_PATH determines where Git looks for its sub-programs (like `git-commit`, `git-diff`, and others). You can check the current setting by running `git --exec-path`.

HOME isn't usually considered customizable (too many other things depend on it), but it's where Git looks for the global configuration file. If you want a truly portable Git installation, complete with global configuration, you can override `HOME` in the portable Git's shell profile.

PREFIX is similar, but for the system-wide configuration. Git looks for this file at `$PREFIX/etc/gitconfig`.

GIT_CONFIG_NOSYSTEM, if set, disables the use of the system-wide configuration file. This is useful if your system config is interfering with your commands, but you don't have access to change or remove it.

GIT_PAGER controls the program used to display multi-page output on the command line. If this is unset, PAGER will be used as a fallback.

GIT_EDITOR is the editor Git will launch when the user needs to edit some text (a commit message, for example). If unset, EDITOR will be used.

Repository Locations

Git uses several environment variables to determine how it interfaces with the current repository.

GIT_DIR is the location of the `.git` folder. If this isn't specified, Git walks up the directory tree until it gets to `~` or `/`, looking for a `.git` directory at every step.

GIT_CEILING_DIRECTORIES controls the behavior of searching for a `.git` directory. If you access directories that are slow to load (such as those on a tape drive, or across a slow network connection), you may want to have Git stop try-

ing earlier than it might otherwise, especially if Git is invoked when building your shell prompt.

GIT_WORK_TREE is the location of the root of the working directory for a non-bare repository. If not specified, the parent directory of \$GIT_DIR is used.

GIT_INDEX_FILE is the path to the index file (non-bare repositories only).

GIT_OBJECT_DIRECTORY can be used to specify the location of the directory that usually resides at .git/objects.

GIT_ALTERNATE_OBJECT_DIRECTORIES is a colon-separated list (formatted like /dir/one:/dir/two:...) which tells Git where to check for objects if they aren't in GIT_OBJECT_DIRECTORY. If you happen to have a lot of projects with large files that have the exact same contents, this can be used to avoid storing too many copies of them.

Pathspecs

A “pathspec” refers to how you specify paths to things in Git, including the use of wildcards. These are used in the .gitignore file, but also on the command-line (git add *.c).

GIT_GLOB_PATHSPECS and **GIT_NOGLOB_PATHSPECS** control the default behavior of wildcards in pathspecs. If GIT_GLOB_PATHSPECS is set to 1, wildcard characters act as wildcards (which is the default); if GIT_NOGLOB_PATHSPECS is set to 1, wildcard characters only match themselves, meaning something like *.c would only match a file named “*.c”, rather than any file whose name ends with .c. You can override this in individual cases by starting the pathspec with :(glob) or :(literal), as in :(glob)*.c.

GIT_LITERAL_PATHSPECS disables both of the above behaviors; no wildcard characters will work, and the override prefixes are disabled as well.

GIT_ICASE_PATHSPECS sets all pathspecs to work in a case-insensitive manner.

Committing

The final creation of a Git commit object is usually done by git-commit-tree, which uses these environment variables as its primary source of information, falling back to configuration values only if these aren't present.

GIT_AUTHOR_NAME is the human-readable name in the “author” field.

GIT_AUTHOR_EMAIL is the email for the “author” field.

GIT_AUTHOR_DATE is the timestamp used for the “author” field.

GIT_COMMITTER_NAME sets the human name for the “committer” field.

GIT_COMMITTER_EMAIL is the email address for the “committer” field.

GIT_COMMITTER_DATE is used for the timestamp in the “committer” field.

EMAIL is the fallback email address in case the `user.email` configuration value isn’t set. If *this* isn’t set, Git falls back to the system user and host names.

Networking

Git uses the `curl` library to do network operations over HTTP, so **GIT_CURL_VERBOSE** tells Git to emit all the messages generated by that library. This is similar to doing `curl -v` on the command line.

GIT_SSL_NO_VERIFY tells Git not to verify SSL certificates. This can sometimes be necessary if you’re using a self-signed certificate to serve Git repositories over HTTPS, or you’re in the middle of setting up a Git server but haven’t installed a full certificate yet.

If the data rate of an HTTP operation is lower than **GIT_HTTP_LOW_SPEED_LIMIT** bytes per second for longer than **GIT_HTTP_LOW_SPEED_TIME** seconds, Git will abort that operation. These values override the `http.lowSpeedLimit` and `http.lowSpeedTime` configuration values.

GIT_HTTP_USER_AGENT sets the user-agent string used by Git when communicating over HTTP. The default is a value like `git/2.0.0`.

Diffing and Merging

GIT_DIFF_OPTS is a bit of a misnomer. The only valid values are `-u<n>` or `--unified=<n>`, which controls the number of context lines shown in a `git diff` command.

GIT_EXTERNAL_DIFF is used as an override for the `diff.external` configuration value. If it’s set, Git will invoke this program when `git diff` is invoked.

GIT_DIFF_PATH_COUNTER and **GIT_DIFF_PATH_TOTAL** are useful from inside the program specified by **GIT_EXTERNAL_DIFF** or `diff.external`. The former represents which file in a series is being diffed (starting with 1), and the latter is the total number of files in the batch.

GIT_MERGE_VERBOSITY controls the output for the recursive merge strategy. The allowed values are as follows:

- 0 outputs nothing, except possibly a single error message.
- 1 shows only conflicts.
- 2 also shows file changes.
- 3 shows when files are skipped because they haven’t changed.
- 4 shows all paths as they are processed.

- 5 and above show detailed debugging information.

The default value is 2.

Debugging

Want to *really* know what Git is up to? Git has a fairly complete set of traces embedded, and all you need to do is turn them on. The possible values of these variables are as follows:

- “true”, “1”, or “2” – the trace category is written to stderr.
- An absolute path starting with / – the trace output will be written to that file.

GIT_TRACE controls general traces, which don’t fit into any specific category. This includes the expansion of aliases, and delegation to other subprograms.

```
$ GIT_TRACE=true git lga
20:12:49.877982 git.c:554
20:12:49.878369 run-command.c:341
20:12:49.879529 git.c:282
20:12:49.879885 git.c:349
20:12:49.899217 run-command.c:341
20:12:49.899675 run-command.c:192
                                         trace: exec: 'git-lga'
                                         trace: run_command: 'git-lga'
                                         trace: alias expansion: lga => 'log' '--graph' '--pr
                                         trace: built-in: git 'log' '--graph' '--pretty=oneli
                                         trace: run_command: 'less'
                                         trace: exec: 'less'
```

GIT_TRACE_PACK_ACCESS controls tracing of packfile access. The first field is the packfile being accessed, the second is the offset within that file:

```
$ GIT_TRACE_PACK_ACCESS=true git status
20:10:12.081397 sha1_file.c:2088      .git/objects/pack/pack-c3fa...291e.pack 12
20:10:12.081886 sha1_file.c:2088      .git/objects/pack/pack-c3fa...291e.pack 34662
20:10:12.082115 sha1_file.c:2088      .git/objects/pack/pack-c3fa...291e.pack 35175
# [...]
20:10:12.087398 sha1_file.c:2088      .git/objects/pack/pack-e80e...e3d2.pack 56914983
20:10:12.087419 sha1_file.c:2088      .git/objects/pack/pack-e80e...e3d2.pack 14303666
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

GIT_TRACE_PACKET enables packet-level tracing for network operations.

```
$ GIT_TRACE_PACKET=true git ls-remote origin
20:15:14.867043 pkt-line.c:46          packet:      git< # service=git-upload
20:15:14.867071 pkt-line.c:46          packet:      git< 0000
20:15:14.867079 pkt-line.c:46          packet:      git< 97b8860c071898d9e162
20:15:14.867088 pkt-line.c:46          packet:      git< 0f20ae29889d61f2e93a
20:15:14.867094 pkt-line.c:46          packet:      git< 36dc827bc9d17f80ed4f
# [...]
```

GIT_TRACE_PERFORMANCE controls logging of performance data. The output shows how long each particular git invocation takes.

```
$ GIT_TRACE_PERFORMANCE=true git gc
20:18:19.499676 trace.c:414           performance: 0.374835000 s: git command: 'gc'
20:18:19.845585 trace.c:414           performance: 0.343020000 s: git command: 'gc'
Counting objects: 170994, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (43413/43413), done.
Writing objects: 100% (170994/170994), done.
Total 170994 (delta 126176), reused 170524 (delta 125706)
20:18:23.567927 trace.c:414           performance: 3.715349000 s: git command: 'gc'
20:18:23.584728 trace.c:414           performance: 0.000910000 s: git command: 'gc'
20:18:23.605218 trace.c:414           performance: 0.017972000 s: git command: 'gc'
20:18:23.606342 trace.c:414           performance: 3.756312000 s: git command: 'gc'
Checking connectivity: 170994, done.
20:18:25.225424 trace.c:414           performance: 1.616423000 s: git command: 'gc'
20:18:25.232403 trace.c:414           performance: 0.001051000 s: git command: 'gc'
20:18:25.233159 trace.c:414           performance: 6.112217000 s: git command: 'gc'
```

GIT_TRACE_SETUP shows information about what Git is discovering about the repository and environment it's interacting with.

```
$ GIT_TRACE_SETUP=true git status
20:19:47.086765 trace.c:315           setup: git_dir: .git
20:19:47.087184 trace.c:316           setup: worktree: /Users/ben/src/git
20:19:47.087191 trace.c:317           setup: cwd: /Users/ben/src/git
20:19:47.087194 trace.c:318           setup: prefix: (null)
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

Miscellaneous

GIT_SSH, if specified, is a program that is invoked instead of ssh when Git tries to connect to an SSH host. It is invoked like `$GIT_SSH [username@]host [-p <port>] <command>`. Note that this isn't the easiest way to customize how ssh is invoked; it won't support extra command-line parameters, so you'd have to write a wrapper script and set `GIT_SSH` to point to it. It's probably easier just to use the `~/.ssh/config` file for that.

GIT_ASKPASS is an override for the `core.askpass` configuration value. This is the program invoked whenever Git needs to ask the user for credentials, which can expect a text prompt as a command-line argument, and should return the answer on `stdout`. (See “**Credential Storage**” for more on this subsystem.)

GIT_NAMESPACE controls access to namespaced refs, and is equivalent to the `--namespace` flag. This is mostly useful on the server side, where you may want to store multiple forks of a single repository in one repository, only keeping the refs separate.

GIT_FLUSH can be used to force Git to use non-buffered I/O when writing incrementally to `stdout`. A value of 1 causes Git to flush more often, a value of 0 causes all output to be buffered. The default value (if this variable is not set) is to choose an appropriate buffering scheme depending on the activity and the output mode.

GIT_REFLOG_ACTION lets you specify the descriptive text written to the reflog. Here's an example:

```
$ GIT_REFLOG_ACTION="my action" git commit --allow-empty -m 'my message'
[master 9e3d55a] my message
$ git reflog -1
9e3d55a HEAD@{0}: my action: my message
```

Summary

You should have a pretty good understanding of what Git does in the background and, to some degree, how it's implemented. This chapter has covered a number of plumbing commands – commands that are lower level and simpler than the porcelain commands you've learned about in the rest of the book. Understanding how Git works at a lower level should make it easier to understand why it's doing what it's doing and also to write your own tools and helping scripts to make your specific workflow work for you.

Git as a content-addressable filesystem is a very powerful tool that you can easily use as more than just a VCS. We hope you can use your newfound knowledge of Git internals to implement your own cool application of this technology and feel more comfortable using Git in more advanced ways.

Git en otros entornos

A

Si has leído hasta aquí todo el libro, seguro que has aprendido un montón de cosas sobre el uso de Git con la línea de comandos. Se puede trabajar con archivos locales, conectar nuestro repositorio con otros repositorios en la red y realizar nuestro trabajo eficientemente con ellos. Aunque las opciones no terminan ahí, Git se utiliza con parte de un ecosistema mayor y un terminal no siempre es la mejor forma de trabajar. Vamos a ver otros tipos de entornos en los que Git resulta muy útil y cómo otras aplicaciones (incluidas las tuyas) pueden trabajar conjuntamente con Git.

Interfaces gráficas

El entorno nativo de Git es la línea de comandos. Sólo desde la línea de comandos se encuentra disponible todo el poder de Git. El texto plano no siempre es la mejor opción para todas las tareas y en ocasiones se necesita una representación visual además, algunos usuarios se siente más cómodo con una interfaz de apuntar y pulsar.

Conviene advertir que los diferentes interfaces están adaptados para diferentes flujos de trabajo. Algunos clientes sólo disponen de un subconjunto adecuadamente seleccionado de toda la funcionalidad de Git para poder atender una forma concreta de trabajar que los autores consideran eficiente. Bajo esta perspectiva, ninguna de estas herramientas puede considerarse “mejor” que otras, simplemente se ajusta mejor a un objetivo prefijado. Además, no hay nada en estos clientes gráficos que no se encuentre ya en el cliente en línea de comandos y, por tanto, la línea de comandos es el modo con el que se consigue la máxima potencia y control cuando se trabaja con repositorios.

gitk y git-gui

Cuando se instala Git, también se instalan sus herramientas gráficas: `gitk` y `git-gui`.

`gitk` es un visor gráfico del histórico. Hay que considerarlo como una interfaz gráfica mejorada sobre `git log` y `git grep`. Es la herramienta que hay que utilizar cuando se quiere encontrar algo que sucedió en el pasado o visualizar el histórico de un proyecto.

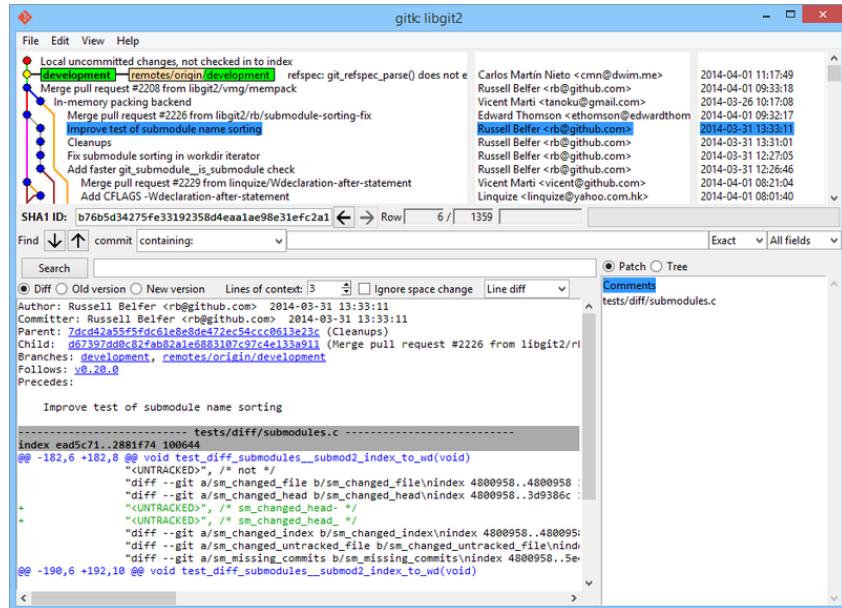
`Gitk` es muy fácil de invocar desde la línea de comandos. Simplemente, hay que moverse con `cd` hasta un repositorio de Git y teclear:

```
$ gitk [git log options]
```

`Gitk` admite muchas opciones desde la línea de comandos, la mayoría de las cuales se pasan desde la acción `git log` subyacente. Probablemente una de las más útiles sea la opción `--all` que indica a `gitk` que muestre todos los commit accesibles desde *cualquier* referencia, no sólo desde el HEAD. La interfaz de `gitk` tiene este aspecto:

Figure 1-1.

El visor de históricos `gitk`.



En la parte superior hay algo que se parece un poco a la salida de `git log --graph`, donde cada punto representa un commit, las líneas representan relaciones entre padres e hijos y las referencias aparecen como cajas coloreadas. El punto amarillo representa una CABEZA o HEAD y el punto rojo a cambios que han sido aceptado con commit. En la parte inferior se encuentra un commit seleccionado con los comentarios y el parche a la izquierda, junto con una vista resumen a la derecha. En medio hay una serie de controles que se usan para buscar en el histórico.

Por su parte, `git-gui` es principalmente una herramienta para elaborar commits. Resulta igual de sencilla de invocar desde la línea de comandos:

```
$ git gui
```

La herramienta tiene este aspecto:

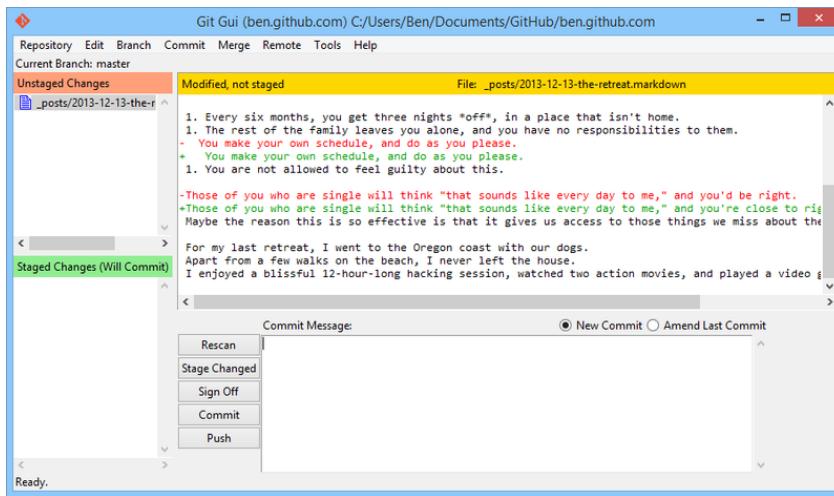


Figure 1-2.

La herramienta `git-gui`.

A la izquierda está el índice con los cambios no preparados al principio y los preparados al final. Es posible mover ficheros completos entre los dos estados haciendo clic en sus respectivos iconos o seleccionar un archivo para visualizarlo haciendo clic en su nombre.

A la derecha, en la parte superior, se encuentra la vista de diferencias que muestra los cambios producidos en el archivo seleccionado. Se pueden preparar bloques o líneas individualmente haciendo clic con el botón derecho sobre esta zona.

En la parte inferior se encuentra el área de mensajes y acciones. Escribiendo un mensaje en la caja de texto y haciendo clic en el botón “Commit” se hace algo lo mismo que con `git commit`. Es posible rectificar el último commit eligiendo ``Amend`` en el botón de opción, con lo que se actualizará la zona de cambios preparados o ``Staged Changes`` con el contenido del último commit. Así es posible poner un cambio como prepreparado o no preparado, modificar el mensaje de un commit y, a continuación, hacer clic en "Commit" para sustituir un commit anterior por uno nuevo.

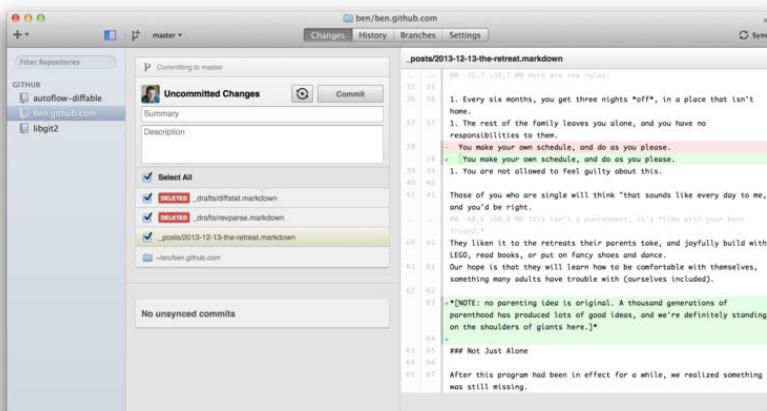
`gitk` y `git-gui` son ejemplos de herramientas orientadas a tareas. Cada una de ellas está adaptada para un propósito específico (visualización del histórico y creación de commits, respectivamente), omitiendo aquellas características que no son necesarias para su tarea.

GitHub para Mac y Windows

GitHub ha creado dos clientes Git: uno para Windows y otro para Mac. Estos clientes son un claro ejemplo de herramientas orientadas a flujo de trabajo, en vez de proporcionar *toda* la funcionalidad de Git, se centran en conjunto conservador de las funcionalidades más utilizadas que colaboran eficazmente. Tienen este aspecto:

Figure 1-3.

GitHub para Mac.



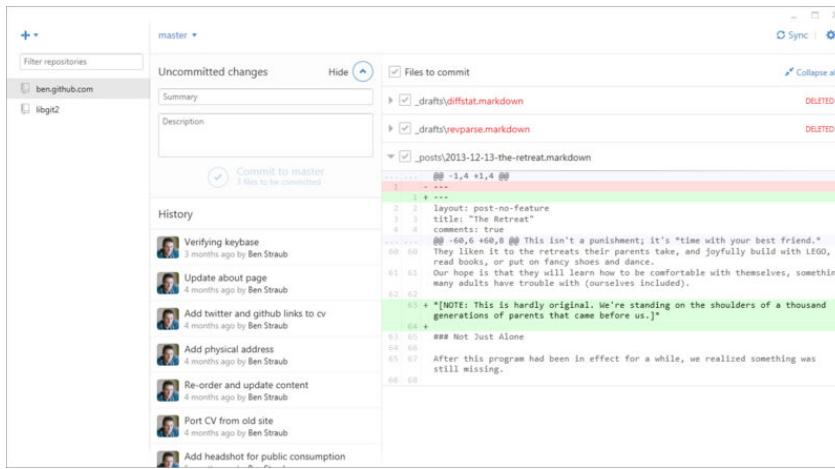


Figure 1-4.

GitHub para Windows.

Se diseñaron para parecer y funcionar de una forma muy similar, por tanto, en este capítulo las trataremos como un mismo producto. Aquí no encontrarás un informe detallado de estas herramientas (para eso tienen su propia documentación), aunque sí hay un recorrido por la vista de “cambios” (que es donde pasaremos la mayor parte del tiempo).

- A la izquierda se encuentra la lista de repositorios que el cliente tiene en seguimiento. Se pueden añadir un repositorio (bien por clonación o agregándolo localmente) pulsado en el ícono “+” que está en la parte superior de este área.
- En el centro está el área de entrada de commit, en la que puedes introducir el mensaje del commit y seleccionar los archivos que se tienen que incluir. (En Windows, se visualiza el histórico de commit directamente debajo mientras que en Mac, lo hace en una pestaña separada.)
- En la parte derecha está el visor de diferencias que muestra los cambios producidos en el directorio de trabajo o los cambios que se incluyeron en el commit seleccionado.
- Por último, se debe tener en cuenta que el botón “Sync”, en la parte superior derecha, es el principal método de interactuar con la red.

Para usar estas herramientas no es necesario tener una cuenta en GitHub. Aunque se diseñaron como punto fuerte del servicio GitHub y del flujo de trabajo recomendado, realmente funcionan con cualquier repositorio y pueden realizar las operaciones de red con cualquier servidor Git.

INSTALACIÓN

GitHub para Windows se puede descargar de <https://windows.github.com> y GitHub para Mac de <https://mac.github.com>. Cuando la aplicación se ejecuta por primera vez, realiza un recorrido por toda la configuración inicial, es decir, se configura el nombre y la dirección de correo electrónico, así como realiza las configuraciones normales para muchas opciones de configuración habituales como la caché de credenciales o el comportamiento del salto de línea (CRLF).

Ambos están “siempre actualizados” - las actualizaciones se descargan e instalan en segundo plano mientras la aplicación está abierta. Incluyen una práctica versión completa de Git que significa que muy probablemente no tengas que preocuparte más de actualizarlo manualmente nunca más. El cliente de Windows incluye un acceso directo para lanzar un Powershell con Posh-git, que veremos más adelante en este capítulo.

El siguiente paso consiste en indicarle a la herramienta algún repositorio para que comience comience a trabajar. El cliente te muestra una lista con los repositorios a los que tienes acceso en GitHub para que puedas clonarlos en un solo paso. Si también dispones de repositorios locales, simplemente arrastra los directorios desde Finder o desde el Explorador de Windows hasta la ventana del cliente Github y se incluirán en la lista de repositorios a la derecha.

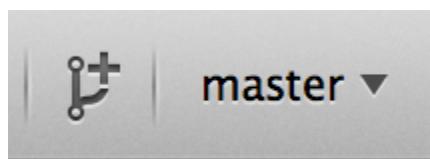
FLUJO DE TRABAJO RECOMENDADO

Una vez instalado y configurado el cliente GitHub está listo para utilizarse en muchas tareas Git habituales. El flujo de trabajo previsto para esta herramienta se denomina “Flujo GitHub.” Veremos con más detalle esto en “[El Flujo de Trabajo en GitHub](#)”, pero, en esencia, consistirá en (a) realizar commits sobre una rama y (b) sincronizarse con un repositorio remoto con cierta regularidad.

La gestión de ramas es uno de los puntos en donde las dos herramientas varían. En Mac, hay un botón en la parte superior de la ventana para crear una rama nueva:

Figure 1-5.

Botón “Create Branch” en Mac.



En Windows, esto se realiza escribiendo el nombre de la nueva rama en la componente de gestión de ramas:

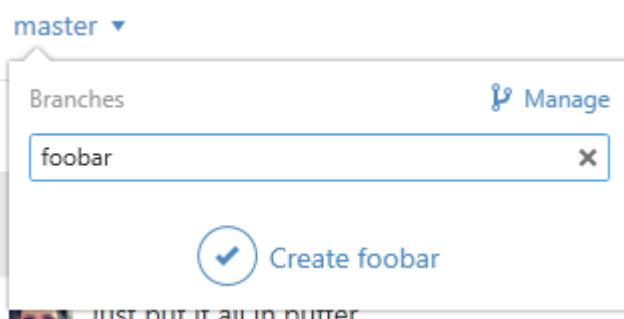


Figure 1-6.

Creación de una rama en Windows.

Una vez creada la rama, realizar nuevos commits sobre ella es bastante sencillo. Realizas algunos cambios en el directorio de trabajo y, si vuelves a la ventana del cliente GitHub, te muestra los archivos que han sido modificados. Introduce un mensaje para el commit, seleccionas los archivos que quieras incluir y pulsas sobre el botón “Commit” (ctrl-enter o ⌘-enter).

La principal forma de interactuar con otros repositorios en la red es mediante la funcionalidad “Sync”. Internamente Git dispone de operaciones diferenciadas para mandar (push), recuperar (fetch), fusionar (merge) y reorganizar (rebase), aunque los clientes GitHub juntan todas ellas en una sola funcionalidad multi-paso. Esto es lo que ocurre cuando se pulsa el botón Sync:

1. `git pull --rebase`. Si esto falla es debido a conflictos en la fusión, echa mano de `git pull --no-rebase`.
1. `git push`.

Esta es la secuencia más frecuente de comandos de red cuando se trabaja de esta manera, así que meter estos comandos ahorra un montón de tiempo.

RESUMEN

Estas herramientas resultan ideales para el flujo de trabajo para el que fueron diseñadas. Tanto desarrolladores como no desarrolladores pueden ponerse a colaborar en un proyecto en cuestión de minutos y la mayoría de las buenas prácticas para esta forma de trabajo vienen integradas con las herramientas. Sin embargo, si el flujo de trabajo es distinto o si se quiere más control sobre cómo y dónde se hacen las operaciones con la red, quizás sea más conveniente algún otro cliente o la línea de comandos.

Otras herramientas gráficas

Existen otra serie de cliente Git con interfaz gráfica que oscilan entre las herramientas especializadas con un único fin y las aplicaciones que intentan presentar todo lo que Git puede hacer. La página oficial de Git tiene una sucinta lista de los clientes más utilizados en <http://git-scm.com/downloads/guis>. Aunque en la wiki de Git se puede encontrar una lista más completa https://git.wiki.kernel.org/index.php/Interfaces,_frontends,_and_tools#Graphical_Interfaces.

Git en Visual Studio

Desde la versión Visual Studio 2013 Update 1, los usuarios de Visual Studio disponen de un cliente Git integrado en el IDE. Visual Studio había tenido funcionalidades integradas de control de versiones desde hace tiempo pero estaban orientadas hacia sistemas centralizados con bloqueo de archivos, así que Git no se adecuaba bien a ese flujo de trabajo. La compatibilidad de Git en Visual Studio 2013 se ha apartado de esta antigua funcionalidad y el resultado es una adaptación mucho mejor entre Visual Studio y Git.

Para localizar esta funcionalidad, abre un proyecto que esté controlado mediante Git (o simplemente usa `git init` en un proyecto ya existente) y selecciona en el menú VIEW > Team Explorer. Puedes ver el visor de “Connect” (Conectar) que se parecerá un poco a ésta:

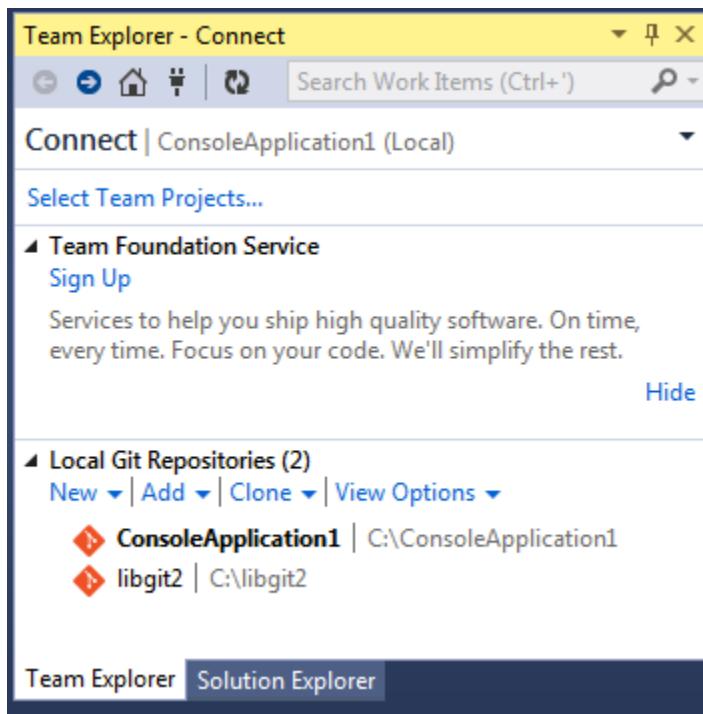


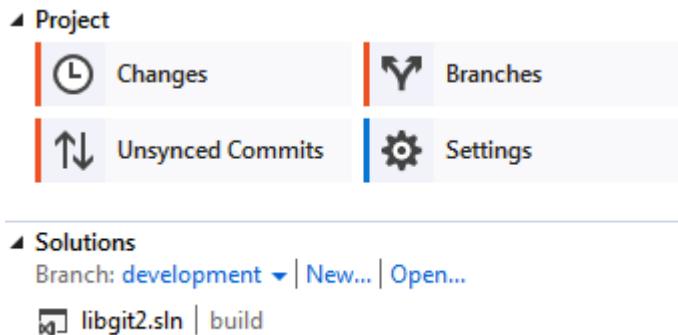
Figure 1-7.

Conectándose a un repositorio Git desde el Team Explorer.

Visual Studio recuerda todos los proyectos que se han abierto y que están controlados mediante Git, y estarán disponibles en la lista de abajo. Si no consigues ver el proyecto, haz clic en el enlace “Add” y escribe la ruta del directorio de trabajo. Haciendo doble clic sobre uno de los repositorios locales Git, te lleva a la vista de inicio, que es como **Figure A-8**. Este es el centro para realizar las acciones Git. Cuando estás *escribiendo* código, probablemente dediques la mayor parte del tiempo sobre el visor de “Changes” (Cambios), aunque cuando llegue el momento de descargar (pull down) los cambios realizados por tus compañeros, seguramente utilizarás los visores de “Unsynced Commits” (Commit no sincronizados) y de “Branches” (Ramas).

Figure 1-8.

Vista de inicio del repositorio Git en Visual Studio.



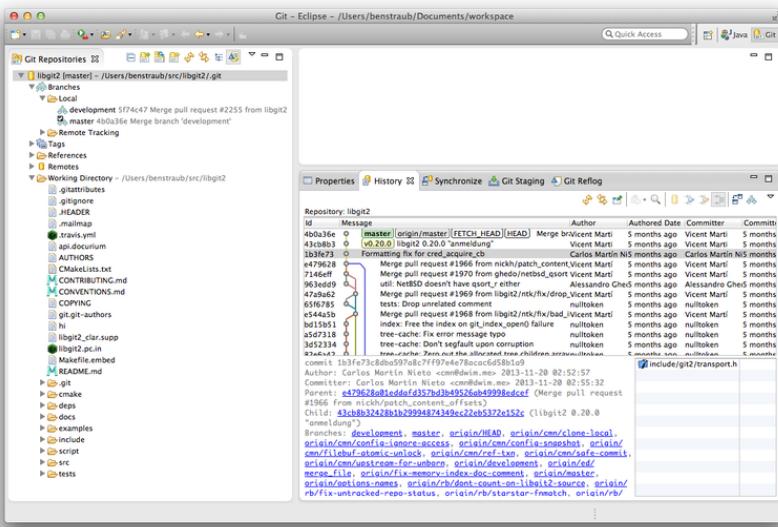
Visual Studio tiene ahora un entorno gráfico para Git potente y orientado a tareas. Incluye un visor de históricos lineal, un visor de diferencias, comandos remotos y otras muchas funcionalidades. Puedes dirigirte a <http://msdn.microsoft.com/en-us/library/hh850437.aspx> para una documentación más completa de todas estas funcionalidades (que no cabrían en esta sección).

Git en Eclipse

Eclipse trae de serie una componente denominada Egit que proporciona una interfaz bastante completa de las operaciones con Git. Para acceder a ella, hay que ir a la perspectiva Git (en el menú Window > Open Perspective > Other..., y entonces seleccionar "Git").

Figure 1-9.

Entorno EGit en Eclipse.



Egit tiene una completa documentación, a la que se puede acceder yendo a Help > Help Contents, y seleccionando el nodo “EGit Documentation” en el listado de contenidos.

Git con Bash

Si eres usuario de Bash, puedes acceder a una serie de características de la consola o terminal que pueden hacer mucho más llevadera la experiencia con Git. Aunque Git viene con extensiones para varios tipos de terminales, éstas no suelen estar activadas por defecto.

Lo primero es obtener una copia del archivo `contrib/completion/git-completion.bash` del código fuente de Git. Haz una copia de este archivo en cualquier lugar, por ejemplo el directorio de inicio, y añádelo al `.bashrc`:

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

Una vez hecho esto, cámbiate a un directorio que sea un repositorio git y teclea:

```
$ git chec<tab>
```

... Bash debería autocompletar con `git checkout`. Esto funciona con todos los subcomandos de Git, parámetros en línea de comandos y en nombres remotos y referencias, cuando sea apropiado.

También resulta útil personalizar el prompt para que muestre información sobre el repositorio Git que hay en el directorio actual. Se puede hacer tan simple o tan complejo como quieras aunque hay una serie de elementos de información que a la mayoría de las personas les resultan útiles, como la rama actual o el estado del directorio de trabajo. Para añadirlo al prompt, simplemente haz una copia del archivo `contrib/completion/git-prompt.sh` del código fuente de Git al directorio de inicio y añade lo siguiente al `.bashrc`:

```
. ~/git-prompt.sh  
export GIT_PS1_SHOWDIRTYSTATE=1  
export PS1='\w$(__git_ps1 " (%s)")\$ '
```

La `\w` indica que muestre el directorio de trabajo actual, la `\$` que muestre el símbolo `$` como parte del prompt y el `__git_ps1 " (%s)"` llama una función en `git-prompt.sh` con un parámetro de formato. Así, el prompt del bash tendrá este aspecto cuando estemos en un proyecto gestionado con Git:

Figure 1-10.

Prompt personalizado en bash.



~/src/libgit2 (development *)\$

Ambos scripts tienen una práctica documentación, por lo que, para más información, revisa los contenidos de `git-completion.bash` y `git-prompt.sh`.

Git in Zsh

Git also ships with a tab-completion library for Zsh. Just copy `contrib/completion/git-completion.zsh` to your home directory and source it from your `.zshrc`. Zsh's interface is a bit more powerful than Bash's:

```
$ git che<tab>
check-attr      -- display gitattributes information
check-ref-format -- ensure that a reference name is well formed
checkout        -- checkout branch or paths to working tree
checkout-index   -- copy files from index to working directory
cherry          -- find commits not merged upstream
cherry-pick     -- apply changes introduced by some existing commits
```

Ambiguous tab-completions aren't just listed; they have helpful descriptions, and you can graphically navigate the list by repeatedly hitting tab. This works with Git commands, their arguments, and names of things inside the repository (like refs and remotes), as well filenames and all the other things Zsh knows how to tab-complete.

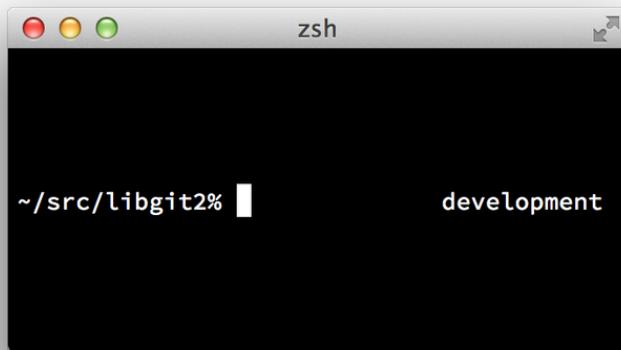
Zsh happens to be fairly compatible with Bash when it comes to prompt customization, but it allows you to have a right-side prompt as well. To include the branch name on the right side, add these lines to your `~/.zshrc` file:

```
setopt prompt_subst
. ~/git-prompt.sh
export RPROMPT='${__git_ps1 "%s"}'
```

This results in a display of the current branch on the right-hand side of the terminal window, whenever your shell is inside a Git repository. It looks a bit like this:

Figure 1-11.

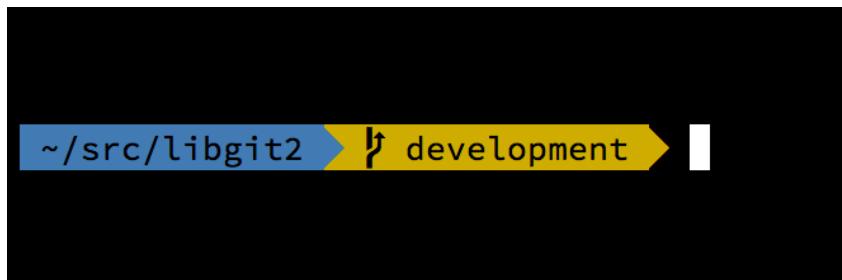
Customized zsh prompt.



Zsh is powerful enough that there are entire frameworks dedicated to making it better. One of them is called “oh-my-zsh”, and it can be found at <https://github.com/robbyrussell/oh-my-zsh>. oh-my-zsh’s plugin system comes with powerful git tab-completion, and it has a variety of prompt “themes”, many of which display version-control data. **Figure A-12** is just one example of what can be done with this system.

Figure 1-12.

An example of an oh-my-zsh theme.



Git in Powershell

The standard command-line terminal on Windows (`cmd.exe`) isn’t really capable of a customized Git experience, but if you’re using Powershell, you’re in luck. A package called Posh-Git (<https://github.com/dahlbyk/posh-git>) pro-

vides powerful tab-completion facilities, as well as an enhanced prompt to help you stay on top of your repository status. It looks like this:

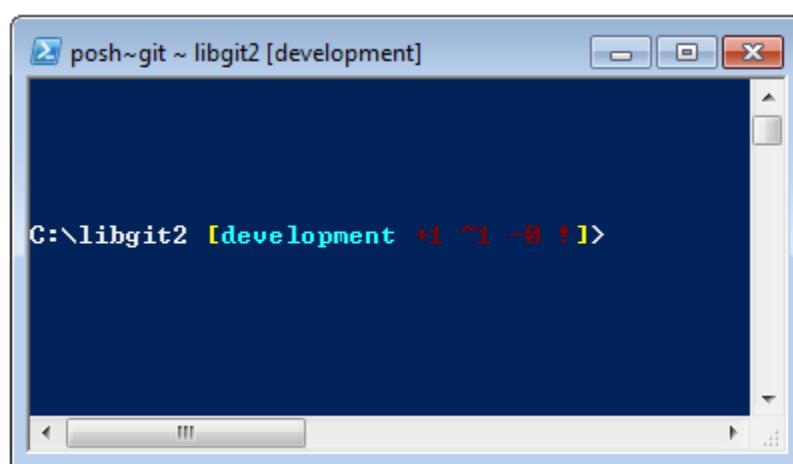


Figure 1-13.

Powershell with Posh-git.

If you've installed GitHub for Windows, Posh-Git is included by default, and all you have to do is add these lines to your `profile.ps1` (which is usually located in `C:\Users\<username>\Documents\WindowsPowerShell`):

```
. (Resolve-Path "$env:LOCALAPPDATA\GitHub\shell.ps1")
. $env:github_posh_git\profile.example.ps1
```

If you're not a GitHub for Windows user, just download a Posh-Git release from (<https://github.com/dahlbyk/posh-git>), and uncompress it to the `WindowsPowerShell` directory. Then open a Powershell prompt as the administrator, and do this:

```
> Set-ExecutionPolicy RemoteSigned -Scope CurrentUser -Confirm
> cd ~\Documents\WindowsPowerShell\posh-git
> .\install.ps1
```

This will add the proper line to your `profile.ps1` file, and posh-git will be active the next time you open your prompt.

Resumen

Has aprendido a sacar partido a toda la potencia de Git desde dentro de la herramienta para poder usarlo en tu trabajo diario así como a acceder a los repositorios Git desde tus propios programas.

Embedding Git in your Applications

B

If your application is for developers, chances are good that it could benefit from integration with source control. Even non-developer applications, such as document editors, could potentially benefit from version-control features, and Git's model works very well for many different scenarios.

If you need to integrate Git with your application, you have essentially three choices: spawning a shell and using the Git command-line tool; Libgit2; and JGit.

Command-line Git

One option is to spawn a shell process and use the Git command-line tool to do the work. This has the benefit of being canonical, and all of Git's features are supported. This also happens to be fairly easy, as most runtime environments have a relatively simple facility for invoking a process with command-line arguments. However, this approach does have some downsides.

One is that all the output is in plain text. This means that you'll have to parse Git's occasionally-changing output format to read progress and result information, which can be inefficient and error-prone.

Another is the lack of error recovery. If a repository is corrupted somehow, or the user has a malformed configuration value, Git will simply refuse to perform many operations.

Yet another is process management. Git requires you to maintain a shell environment on a separate process, which can add unwanted complexity. Trying to coordinate many of these processes (especially when potentially accessing the same repository from several processes) can be quite a challenge.

Libgit2

© Another option at your disposal is to use Libgit2. Libgit2 is a dependency-free implementation of Git, with a focus on having a nice API for use within other programs. You can find it at <http://libgit2.github.com>.

First, let's take a look at what the C API looks like. Here's a whirlwind tour:

```
// Open a repository
git_repository *repo;
int error = git_repository_open(&repo, "/path/to/repository");

// Dereference HEAD to a commit
git_object *head_commit;
error = git_revparse_single(&head_commit, repo, "HEAD^{commit}");
git_commit *commit = (git_commit*)head_commit;

// Print some of the commit's properties
printf("%s", git_commit_message(commit));
const git_signature *author = git_commit_author(commit);
printf("%s <%s>\n", author->name, author->email);
const git_oid *tree_id = git_commit_tree_id(commit);

// Cleanup
git_commit_free(commit);
git_repository_free(repo);
```

The first couple of lines open a Git repository. The `git_repository` type represents a handle to a repository with a cache in memory. This is the simplest method, for when you know the exact path to a repository's working directory or `.git` folder. There's also the `git_repository_open_ext` which includes options for searching, `git_clone` and friends for making a local clone of a remote repository, and `git_repository_init` for creating an entirely new repository.

The second chunk of code uses rev-parse syntax (see “[Branch References](#)” for more on this) to get the commit that HEAD eventually points to. The type returned is a `git_object` pointer, which represents something that exists in the Git object database for a repository. `git_object` is actually a “parent” type for several different kinds of objects; the memory layout for each of the “child” types is the same as for `git_object`, so you can safely cast to the right one. In this case, `git_object_type(commit)` would return `GIT_OBJ_COMMIT`, so it's safe to cast to a `git_commit` pointer.

The next chunk shows how to access the commit's properties. The last line here uses a `git_oid` type; this is Libgit2's representation for a SHA-1 hash.

From this sample, a couple of patterns have started to emerge:

- If you declare a pointer and pass a reference to it into a Libgit2 call, that call will probably return an integer error code. A 0 value indicates success; anything less is an error.
- If Libgit2 populates a pointer for you, you’re responsible for freeing it.
- If Libgit2 returns a `const` pointer from a call, you don’t have to free it, but it will become invalid when the object it belongs to is freed.
- Writing C is a bit painful.

That last one means it isn’t very probable that you’ll be writing C when using Libgit2. Fortunately, there are a number of language-specific bindings available that make it fairly easy to work with Git repositories from your specific language and environment. Let’s take a look at the above example written using the Ruby bindings for Libgit2, which are named Rugged, and can be found at <https://github.com/libgit2/rugged>.

```
repo = Rugged::Repository.new('path/to/repository')
commit = repo.head.target
puts commit.message
puts "#{commit.author[:name]} <#{commit.author[:email]}>"
tree = commit.tree
```

As you can see, the code is much less cluttered. Firstly, Rugged uses exceptions; it can raise things like `ConfigError` or `ObjectError` to signal error conditions. Secondly, there’s no explicit freeing of resources, since Ruby is garbage-collected. Let’s take a look at a slightly more complicated example: crafting a commit from scratch

```
blob_id = repo.write("Blob contents", :blob) ❶

index = repo.index
index.read_tree(repo.head.target.tree)
index.add(:path => 'newfile.txt', :oid => blob_id) ❷

sig = {
  :email => "bob@example.com",
  :name => "Bob User",
  :time => Time.now,
}

commit_id = Rugged::Commit.create(repo,
  :tree => index.write_tree(repo), ❸
  :author => sig,
  :committer => sig, ❹
  :message => "Add newfile.txt", ❺
  :parents => repo.empty? ? [] : [repo.head.target].compact, ❻
  :update_ref => 'HEAD', ❼
```

```
)  
commit = repo.lookup(commit_id) ❸
```

- ❶ Create a new blob, which contains the contents of a new file.
- ❷ Populate the index with the head commit’s tree, and add the new file at the path `newfile.txt`.
- ❸ This creates a new tree in the ODB, and uses it for the new commit.
- ❹ We use the same signature for both the author and committer fields.
- ❺ The commit message.
- ❻ When creating a commit, you have to specify the new commit’s parents. This uses the tip of HEAD for the single parent.
- ❼ Rugged (and Libgit2) can optionally update a reference when making a commit.
- ❽ The return value is the SHA-1 hash of a new commit object, which you can then use to get a `Commit` object.

The Ruby code is nice and clean, but since Libgit2 is doing the heavy lifting, this code will run pretty fast, too. If you’re not a rubyist, we touch on some other bindings in “[Other Bindings](#)”.

Advanced Functionality

Libgit2 has a couple of capabilities that are outside the scope of core Git. One example is pluggability: Libgit2 allows you to provide custom “backends” for several types of operation, so you can store things in a different way than stock Git does. Libgit2 allows custom backends for configuration, ref storage, and the object database, among other things.

Let’s take a look at how this works. The code below is borrowed from the set of backend examples provided by the Libgit2 team (which can be found at <https://github.com/libgit2/libgit2-backends>). Here’s how a custom backend for the object database is set up:

```
git_odb *odb;  
int error = git_odb_new(&odb); ❶  
  
git_odb_backend *my_backend;  
error = git_odb_backend_mine(&my_backend, /*...*/); ❷
```

```

error = git_odb_add_backend(odb, my_backend, 1); ③

git_repository *repo;
error = git_repository_open(&repo, "some-path");
error = git_repository_set_odb(odb); ④

```

(Note that errors are captured, but not handled. We hope your code is better than ours.)

- ❶ Initialize an empty object database (ODB) “frontend,” which will act as a container for the “backends” which are the ones doing the real work.
- ❷ Initialize a custom ODB backend.
- ❸ Add the backend to the frontend.
- ❹ Open a repository, and set it to use our ODB to look up objects.

But what is this `git_odb_backend_mine` thing? Well, that’s the constructor for your own ODB implementation, and you can do whatever you want in there, so long as you fill in the `git_odb_backend` structure properly. Here’s what it *could* look like:

```

typedef struct {
    git_odb_backend parent;

    // Some other stuff
    void *custom_context;
} my_backend_struct;

int git_odb_backend_mine(git_odb_backend **backend_out, /*...*/)
{
    my_backend_struct *backend;

    backend = calloc(1, sizeof (my_backend_struct));

    backend->custom_context = ...;

    backend->parent.read = &my_backend_read;
    backend->parent.read_prefix = &my_backend_read_prefix;
    backend->parent.read_header = &my_backend_read_header;
    // ...

    *backend_out = (git_odb_backend *) backend;

    return GIT_SUCCESS;
}

```

The subtlest constraint here is that `my_backend_struct`'s first member must be a `git_odb_backend` structure; this ensures that the memory layout is what the Libgit2 code expects it to be. The rest of it is arbitrary; this structure can be as large or small as you need it to be.

The initialization function allocates some memory for the structure, sets up the custom context, and then fills in the members of the parent structure that it supports. Take a look at the `include/git2/sys/odb_backend.h` file in the Libgit2 source for a complete set of call signatures; your particular use case will help determine which of these you'll want to support.

Other Bindings

Libgit2 has bindings for many languages. Here we show a small example using a few of the more complete bindings packages as of this writing; libraries exist for many other languages, including C++, Go, Node.js, Erlang, and the JVM, all in various stages of maturity. The official collection of bindings can be found by browsing the repositories at <https://github.com/libgit2>. The code we'll write will return the commit message from the commit eventually pointed to by HEAD (sort of like `git log -1`).

LIBGIT2SHARP

If you're writing a .NET or Mono application, LibGit2Sharp (<https://github.com/libgit2/libgit2sharp>) is what you're looking for. The bindings are written in C#, and great care has been taken to wrap the raw Libgit2 calls with native-feeling CLR APIs. Here's what our example program looks like:

```
new Repository(@"C:\path\to\repo").Head.Tip.Message;
```

For desktop Windows applications, there's even a NuGet package that will help you get started quickly.

OBJECTIVE-GIT

If your application is running on an Apple platform, you're likely using Objective-C as your implementation language. Objective-Git (<https://github.com/libgit2/objective-git>) is the name of the Libgit2 bindings for that environment. The example program looks like this:

```
GTRepository *repo =
    [[GTRepository alloc] initWithURL:[NSURL fileURLWithPath: @"/path/to/repo"] error:&error];
NSString *msg = [[[repo headReferenceWithError:NULL] resolvedTarget] message];
```

Objective-git is fully interoperable with Swift, so don't fear if you've left Objective-C behind.

PYGIT2

The bindings for Libgit2 in Python are called Pygit2, and can be found at <http://www.pygit2.org/>. Our example program:

```
pygit2.Repository("/path/to/repo") # open repository
    .head                      # get the current branch
    .peel(pygit2.Commit)        # walk down to the commit
    .message                    # read the message
```

Further Reading

Of course, a full treatment of Libgit2's capabilities is outside the scope of this book. If you want more information on Libgit2 itself, there's API documentation at <https://libgit2.github.com/libgit2>, and a set of guides at <https://libgit2.github.com/docs>. For the other bindings, check the bundled README and tests; there are often small tutorials and pointers to further reading there.

JGit

If you want to use Git from within a Java program, there is a fully featured Git library called JGit. JGit is a relatively full-featured implementation of Git written natively in Java, and is widely used in the Java community. The JGit project is under the Eclipse umbrella, and its home can be found at <http://www.eclipse.org/jgit>.

Getting Set Up

There are a number of ways to connect your project with JGit and start writing code against it. Probably the easiest is to use Maven – the integration is accomplished by adding the following snippet to the <dependencies> tag in your pom.xml file:

```
<dependency>
    <groupId>org.eclipse.jgit</groupId>
    <artifactId>org.eclipse.jgit</artifactId>
    <version>3.5.0.201409260305-r</version>
</dependency>
```

The version will most likely have advanced by the time you read this; check <http://mvnrepository.com/artifact/org.eclipse.jgit/org.eclipse.jgit> for updated repository information. Once this step is done, Maven will automatically acquire and use the JGit libraries that you'll need.

If you would rather manage the binary dependencies yourself, pre-built JGit binaries are available from <http://www.eclipse.org/jgit/download>. You can build them into your project by running a command like this:

```
javac -cp .:org.eclipse.jgit-3.5.0.201409260305-r.jar App.java  
java -cp .:org.eclipse.jgit-3.5.0.201409260305-r.jar App
```

Plumbing

JGit has two basic levels of API: plumbing and porcelain. The terminology for these comes from Git itself, and JGit is divided into roughly the same kinds of areas: porcelain APIs are a friendly front-end for common user-level actions (the sorts of things a normal user would use the Git command-line tool for), while the plumbing APIs are for interacting with low-level repository objects directly.

The starting point for most JGit sessions is the `Repository` class, and the first thing you'll want to do is create an instance of it. For a filesystem-based repository (yes, JGit allows for other storage models), this is accomplished using `FileRepositoryBuilder`:

```
// Create a new repository; the path must exist  
Repository newlyCreatedRepo = FileRepositoryBuilder.create(  
    new File("/tmp/new_repo/.git"));  
  
// Open an existing repository  
Repository existingRepo = new FileRepositoryBuilder()  
    .setGitDir(new File("my_repo/.git"))  
    .build();
```

The builder has a fluent API for providing all the things it needs to find a Git repository, whether or not your program knows exactly where it's located. It can use environment variables (`.readEnvironment()`), start from a place in the working directory and search (`.setWorkTree(...).findGitDir()`), or just open a known `.git` directory as above.

Once you have a `Repository` instance, you can do all sorts of things with it. Here's a quick sampling:

```

// Get a reference
Ref master = repo.getRef("master");

// Get the object the reference points to
ObjectId masterTip = master.getObjectId();

// Rev-parse
ObjectId obj = repo.resolve("HEAD^{tree}");

// Load raw object contents
ObjectLoader loader = repo.open(masterTip);
loader.copyTo(System.out);

// Create a branch
RefUpdate createBranch1 = repo.updateRef("refs/heads/branch1");
createBranch1.setNewObjectId(masterTip);
createBranch1.update();

// Delete a branch
RefUpdate deleteBranch1 = repo.updateRef("refs/heads/branch1");
deleteBranch1.setForceUpdate(true);
deleteBranch1.delete();

// Config
Config cfg = repo.getConfig();
String name = cfg.getString("user", null, "name");

```

There's quite a bit going on here, so let's go through it one section at a time.

The first line gets a pointer to the `master` reference. JGit automatically grabs the *actual* master ref, which lives at `refs/heads/master`, and returns an object that lets you fetch information about the reference. You can get the name (`.getName()`), and either the target object of a direct reference (`.getObjectId()`) or the reference pointed to by a symbolic ref (`.getTarget()`). Ref objects are also used to represent tag refs and objects, so you can ask if the tag is "peeled," meaning that it points to the final target of a (potentially long) string of tag objects.

The second line gets the target of the `master` reference, which is returned as an `ObjectId` instance. `ObjectId` represents the SHA-1 hash of an object, which might or might not exist in Git's object database. The third line is similar, but shows how JGit handles the rev-parse syntax (for more on this, see "["Branch References"](#)); you can pass any object specifier that Git understands, and JGit will return either a valid `ObjectId` for that object, or `null`.

The next two lines show how to load the raw contents of an object. In this example, we call `ObjectLoader.copyTo()` to stream the contents of the object directly to stdout, but `ObjectLoader` also has methods to read the type and size of an object, as well as return it as a byte array. For large objects

(where `.isLarge()` returns `true`), you can call `.openStream()` to get an `InputStream`-like object that can read the raw object data without pulling it all into memory at once.

The next few lines show what it takes to create a new branch. We create a `RefUpdate` instance, configure some parameters, and call `.update()` to trigger the change. Directly following this is the code to delete that same branch. Note that `.setForceUpdate(true)` is required for this to work; otherwise the `.delete()` call will return `REJECTED`, and nothing will happen.

The last example shows how to fetch the `user.name` value from the Git configuration files. This `Config` instance uses the repository we opened earlier for local configuration, but will automatically detect the global and system configuration files and read values from them as well.

This is only a small sampling of the full plumbing API; there are many more methods and classes available. Also not shown here is the way JGit handles errors, which is through the use of exceptions. JGit APIs sometimes throw standard Java exceptions (such as `IOException`), but there are a host of JGit-specific exception types that are provided as well (such as `NoRemoteRepositoryException`, `CorruptObjectException`, and `NoMergeBaseException`).

Porcelain

The plumbing APIs are rather complete, but it can be cumbersome to string them together to achieve common goals, like adding a file to the index, or making a new commit. JGit provides a higher-level set of APIs to help out with this, and the entry point to these APIs is the `Git` class:

```
Repository repo;  
// construct repo...  
Git git = new Git(repo);
```

The `Git` class has a nice set of high-level *builder-style* methods that can be used to construct some pretty complex behavior. Let's take a look at an example – doing something like `git ls-remote`:

```
CredentialsProvider cp = new UsernamePasswordCredentialsProvider("username", "p4ssw0rd");  
Collection<Ref> remoteRefs = git.lsRemote()  
    .setCredentialsProvider(cp)  
    .setRemote("origin")  
    .setTags(true)  
    .setHeads(false)  
    .call();  
for (Ref ref : remoteRefs) {
```

```
        System.out.println(ref.getName() + " -> " + ref.getObjectId().name());
    }
```

This is a common pattern with the Git class; the methods return a command object that lets you chain method calls to set parameters, which are executed when you call `.call()`. In this case, we're asking the `origin` remote for tags, but not heads. Also notice the use of a `CredentialsProvider` object for authentication.

Many other commands are available through the Git class, including but not limited to `add`, `blame`, `commit`, `clean`, `push`, `rebase`, `revert`, and `reset`.

Further Reading

This is only a small sampling of JGit's full capabilities. If you're interested and want to learn more, here's where to look for information and inspiration:

- The official JGit API documentation is available online at <http://download.eclipse.org/jgit/docs/latest/apidocs>. These are standard Javadoc, so your favorite JVM IDE will be able to install them locally, as well.
- The JGit Cookbook at <https://github.com/centic9/jgit-cookbook> has many examples of how to do specific tasks with JGit.
- There are several good resources pointed out at <http://stackoverflow.com/questions/6861881>.

Git Commands



Throughout the book we have introduced dozens of Git commands and have tried hard to introduce them within something of a narrative, adding more commands to the story slowly. However, this leaves us with examples of usage of the commands somewhat scattered throughout the whole book.

In this appendix, we'll go through all the Git commands we addressed throughout the book, grouped roughly by what they're used for. We'll talk about what each command very generally does and then point out where in the book you can find us having used it.

Setup and Config

There are two commands that are used quite a lot, from the first invocations of Git to common every day tweaking and referencing, the `config` and `help` commands.

`git config`

Git has a default way of doing hundreds of things. For a lot of these things, you can tell Git to default to doing them a different way, or set your preferences. This involves everything from telling Git what your name is to specific terminal color preferences or what editor you use. There are several files this command will read from and write to so you can set values globally or down to specific repositories.

The `git config` command has been used in nearly every chapter of the book.

In “[Configurando Git por primera vez](#)” we used it to specify our name, email address and editor preference before we even got started using Git.

In “[Alias de Git](#)” we showed how you could use it to create shorthand commands that expand to long option sequences so you don't have to type them every time.

In “**Reorganizar el Trabajo Realizado**” we used it to make --rebase the default when you run `git pull`.

In “**Credential Storage**” we used it to set up a default store for your HTTP passwords.

In “**Expansión de palabras clave**” we showed how to set up smudge and clean filters on content coming in and out of Git.

Finally, basically the entirety of “**Configuración de Git**” is dedicated to the command.

git help

The `git help` command is used to show you all the documentation shipped with Git about any command. While we’re giving a rough overview of most of the more popular ones in this appendix, for a full listing of all of the possible options and flags for every command, you can always run `git help <command>`.

We introduced the `git help` command in “[¿Cómo obtener ayuda?](#)” and showed you how to use it to find more information about the `git shell` in “[Configurando el servidor](#)”.

Getting and Creating Projects

There are two ways to get a Git repository. One is to copy it from an existing repository on the network or elsewhere and the other is to create a new one in an existing directory.

git init

To take a directory and turn it into a new Git repository so you can start version controlling it, you can simply run `git init`.

We first introduce this in “[Obteniendo un repositorio Git](#)”, where we show creating a brand new repository to start working with.

We talk briefly about how you can change the default branch from “master” in “[Ramas Remotas](#)”.

We use this command to create an empty bare repository for a server in “[Colocando un Repositorio Vacío en un Servidor](#)”.

Finally, we go through some of the details of what it actually does behind the scenes in “[Plumbing and Porcelain](#)”.

git clone

The `git clone` command is actually something of a wrapper around several other commands. It creates a new directory, goes into it and runs `git init` to make it an empty Git repository, adds a remote (`git remote add`) to the URL that you pass it (by default named `origin`), runs a `git fetch` from that remote repository and then checks out the latest commit into your working directory with `git checkout`.

The `git clone` command is used in dozens of places throughout the book, but we'll just list a few interesting places.

It's basically introduced and explained in “**Clonando un repositorio existente**”, where we go through a few examples.

In “**Configurando Git en un servidor**” we look at using the `--bare` option to create a copy of a Git repository with no working directory.

In “**Bundling**” we use it to unbundle a bundled Git repository.

Finally, in “**Cloning a Project with Submodules**” we learn the `--recursive` option to make cloning a repository with submodules a little simpler.

Though it's used in many other places throughout the book, these are the ones that are somewhat unique or where it is used in ways that are a little different.

Basic Snapshotting

For the basic workflow of staging content and committing it to your history, there are only a few basic commands.

git add

The `git add` command adds content from the working directory into the staging area (or “index”) for the next commit. When the `git commit` command is run, by default it only looks at this staging area, so `git add` is used to craft what exactly you would like your next commit snapshot to look like.

This command is an incredibly important command in Git and is mentioned or used dozens of times in this book. We'll quickly cover some of the unique uses that can be found.

We first introduce and explain `git add` in detail in “**Rastrear Archivos Nuevos**”.

We mention how to use it to resolve merge conflicts in “**Principales Conflictos que Pueden Surgir en las Fusiones**”.

We go over using it to interactively stage only specific parts of a modified file in “**Interactive Staging**”.

Finally, we emulate it at a low level in “**Tree Objects**”, so you can get an idea of what it’s doing behind the scenes.

git status

The `git status` command will show you the different states of files in your working directory and staging area. Which files are modified and unstaged and which are staged but not yet committed. In its normal form, it also will show you some basic hints on how to move files between these stages.

We first cover `status` in “**Revisando el Estado de tus Archivos**”, both in its basic and simplified forms. While we use it throughout the book, pretty much everything you can do with the `git status` command is covered there.

git diff

The `git diff` command is used when you want to see differences between any two trees. This could be the difference between your working environment and your staging area (`git diff` by itself), between your staging area and your last commit (`git diff --staged`), or between two commits (`git diff master branchB`).

We first look at the basic uses of `git diff` in “**Ver los Cambios Preparados y No Preparados**”, where we show how to see what changes are staged and which are not yet staged.

We use it to look for possible whitespace issues before committing with the `--check` option in “**Commit Guidelines**”.

We see how to check the differences between branches more effectively with the `git diff A...B` syntax in “**Decidiendo qué introducir**”.

We use it to filter out whitespace differences with `-w` and how to compare different stages of conflicted files with `--theirs`, `--ours` and `--base` in “**Advanced Merging**”.

Finally, we use it to effectively compare submodule changes with `--submodule` in “**Starting with Submodules**”.

git difftool

The `git difftool` command simply launches an external tool to show you the difference between two trees in case you want to use something other than the built in `git diff` command.

We only briefly mention this in “[Ver los Cambios Preparados y No Preparados](#)”.

git commit

The `git commit` command takes all the file contents that have been staged with `git add` and records a new permanent snapshot in the database and then moves the branch pointer on the current branch up to it.

We first cover the basics of committing in “[Confirmar tus Cambios](#)”. There we also demonstrate how to use the `-a` flag to skip the `git add` step in daily workflows and how to use the `-m` flag to pass a commit message in on the command line instead of firing up an editor.

In “[Deshacer Cosas](#)” we cover using the `--amend` option to redo the most recent commit.

In “[¿Qué es una rama?](#)”, we go into much more detail about what `git commit` does and why it does it like that.

We looked at how to sign commits cryptographically with the `-S` flag in “[Signing Commits](#)”.

Finally, we take a look at what the `git commit` command does in the background and how it's actually implemented in “[Commit Objects](#)”.

git reset

The `git reset` command is primarily used to undo things, as you can possibly tell by the verb. It moves around the HEAD pointer and optionally changes the index or staging area and can also optionally change the working directory if you use `--hard`. This final option makes it possible for this command to lose your work if used incorrectly, so make sure you understand it before using it.

We first effectively cover the simplest use of `git reset` in “[Deshacer un Archivo Preparado](#)”, where we use it to unstage a file we had run `git add` on.

We then cover it in quite some detail in “[Reset Demystified](#)”, which is entirely devoted to explaining this command.

We use `git reset --hard` to abort a merge in “[Aborting a Merge](#)”, where we also use `git merge --abort`, which is a bit of a wrapper for the `git reset` command.

git rm

The `git rm` command is used to remove files from the staging area and working directory for Git. It is similar to `git add` in that it stages a removal of a file for the next commit.

We cover the `git rm` command in some detail in “[Eliminar Archivos](#)”, including recursively removing files and only removing files from the staging area but leaving them in the working directory with `--cached`.

The only other differing use of `git rm` in the book is in “[Removing Objects](#)” where we briefly use and explain the `--ignore-unmatch` when running `git filter-branch`, which simply makes it not error out when the file we are trying to remove doesn’t exist. This can be useful for scripting purposes.

git mv

The `git mv` command is a thin convenience command to move a file and then run `git add` on the new file and `git rm` on the old file.

We only briefly mention this command in “[Cambiar el Nombre de los Archivos](#)”.

git clean

The `git clean` command is used to remove unwanted files from your working directory. This could include removing temporary build artifacts or merge conflict files.

We cover many of the options and scenarios in which you might use the `clean` command in “[Cleaning your Working Directory](#)”.

Branching and Merging

There are just a handful of commands that implement most of the branching and merging functionality in Git.

git branch

The `git branch` command is actually something of a branch management tool. It can list the branches you have, create a new branch, delete branches and rename branches.

Most of [Chapter 3](#) is dedicated to the `branch` command and it’s used throughout the entire chapter. We first introduce it in “[Crear una Rama Nue-](#)

va” and we go through most of it’s other features (listing and deleting) in “**Ges-tión de Ramas**”.

In “**Hacer Seguimiento a las Ramas**” we use the `git branch -u` option to set up a tracking branch.

Finally, we go through some of what it does in the background in “**Git References**”.

git checkout

The `git checkout` command is used to switch branches and check content out into your working directory.

We first encounter the command in “**Cambiar de Rama**” along with the `git branch` command.

We see how to use it to start tracking branches with the `--track` flag in “**Hacer Seguimiento a las Ramas**”.

We use it to reintroduce file conflicts with `--conflict=diff3` in “**Checking Out Conflicts**”.

We go into closer detail on it’s relationship with `git reset` in “**Reset Demystified**”.

Finally, we go into some implementation detail in “**The HEAD**”.

git merge

The `git merge` tool is used to merge one or more branches into the branch you have checked out. It will then advance the current branch to the result of the merge.

The `git merge` command was first introduced in “**Procedimientos Básicos de Ramificación**”. Though it is used in various places in the book, there are very few variations of the `merge` command — generally just `git merge <branch>` with the name of the single branch you want to merge in.

We covered how to do a squashed merge (where Git merges the work but pretends like it’s just a new commit without recording the history of the branch you’re merging in) at the very end of “**Forked Public Project**”.

We went over a lot about the merge process and command, including the `-Xignore-all-whitespace` command and the `--abort` flag to abort a problem merge in “**Advanced Merging**”.

We learned how to verify signatures before merging if your project is using GPG signing in “**Siging Commits**”.

Finally, we learned about Subtree merging in “**Subtree Merging**”.

git mergetool

The `git mergetool` command simply launches an external merge helper in case you have issues with a merge in Git.

We mention it quickly in “**Principales Conflictos que Pueden Surgir en las Fusiones**” and go into detail on how to implement your own external merge tool in “**Herramientas externas para fusión y diferencias**”.

git log

The `git log` command is used to show the reachable recorded history of a project from the most recent commit snapshot backwards. By default it will only show the history of the branch you’re currently on, but can be given different or even multiple heads or branches from which to traverse. It is also often used to show differences between two or more branches at the commit level.

This command is used in nearly every chapter of the book to demonstrate the history of a project.

We introduce the command and cover it in some depth in “**Ver el Historial de Confirmaciones**”. There we look at the `-p` and `--stat` option to get an idea of what was introduced in each commit and the `--pretty` and `--oneline` options to view the history more concisely, along with some simple date and author filtering options.

In “**Crear una Rama Nueva**” we use it with the `--decorate` option to easily visualize where our branch pointers are located and we also use the `--graph` option to see what divergent histories look like.

In “**Private Small Team**” and “**Commit Ranges**” we cover the `branchA..branchB` syntax to use the `git log` command to see what commits are unique to a branch relative to another branch. In “**Commit Ranges**” we go through this fairly extensively.

In “**Merge Log**” and “**Triple Dot**” we cover using the `branchA...branchB` format and the `--left-right` syntax to see what is in one branch or the other but not in both. In “**Merge Log**” we also look at how to use the `--merge` option to help with merge conflict debugging as well as using the `--cc` option to look at merge commit conflicts in your history.

In “**RefLog Shortnames**” we use the `-g` option to view the Git reflog through this tool instead of doing branch traversal.

In “**Searching**” we look at using the `-S` and `-L` options to do fairly sophisticated searches for something that happened historically in the code such as seeing the history of a function.

In “[Signing Commits](#)” we see how to use `--show-signature` to add a validation string to each commit in the `git log` output based on if it was validly signed or not.

git stash

The `git stash` command is used to temporarily store uncommitted work in order to clean out your working directory without having to commit unfinished work on a branch.

This is basically entirely covered in “[Stashing and Cleaning](#)”.

git tag

The `git tag` command is used to give a permanent bookmark to a specific point in the code history. Generally this is used for things like releases.

This command is introduced and covered in detail in “[Etiquetado](#)” and we use it in practice in “[Etiquetando tus versiones](#)”.

We also cover how to create a GPG signed tag with the `-s` flag and verify one with the `-v` flag in “[Signing Your Work](#)”.

Sharing and Updating Projects

There are not very many commands in Git that access the network, nearly all of the commands operate on the local database. When you are ready to share your work or pull changes from elsewhere, there are a handful of commands that deal with remote repositories.

git fetch

The `git fetch` command communicates with a remote repository and fetches down all the information that is in that repository that is not in your current one and stores it in your local database.

We first look at this command in “[Traer y Combinar Remotos](#)” and we continue to see examples of its use in “[Ramas Remotas](#)”.

We also use it in several of the examples in “[Contributing to a Project](#)”.

We use it to fetch a single specific reference that is outside of the default space in “[Referencias de Pull Request](#)” and we see how to fetch from a bundle in “[Bundling](#)”.

We set up highly custom refspecs in order to make `git fetch` do something a little different than the default in “[The Refspec](#)”.

git pull

The `git pull` command is basically a combination of the `git fetch` and `git merge` commands, where Git will fetch from the remote you specify and then immediately try to merge it into the branch you're on.

We introduce it quickly in “[Traer y Combinar Remotos](#)” and show how to see what it will merge if you run it in “[Inspeccionar un Remoto](#)”.

We also see how to use it to help with rebasing difficulties in “[Reorganizar una Reorganización](#)”.

We show how to use it with a URL to pull in changes in a one-off fashion in “[Recuperando ramas remotas](#)”.

Finally, we very quickly mention that you can use the `--verify-signatures` option to it in order to verify that commits you are pulling have been GPG signed in “[Signing Commits](#)”.

git push

The `git push` command is used to communicate with another repository, calculate what your local database has that the remote one does not, and then pushes the difference into the other repository. It requires write access to the other repository and so normally is authenticated somehow.

We first look at the `git push` command in “[Enviar a Tus Remotos](#)”. Here we cover the basics of pushing a branch to a remote repository. In “[Publicar](#)” we go a little deeper into pushing specific branches and in “[Hacer Seguimiento a las Ramas](#)” we see how to set up tracking branches to automatically push to. In “[Eliminar Ramas Remotas](#)” we use the `--delete` flag to delete a branch on the server with `git push`.

Throughout “[Contributing to a Project](#)” we see several examples of using `git push` to share work on branches through multiple remotes.

We see how to use it to share tags that you have made with the `--tags` option in “[Compartir Etiquetas](#)”.

In “[Publishing Submodule Changes](#)” we use the `--recurse-submodules` option to check that all of our submodules work has been published before pushing the superproject, which can be really helpful when using submodules.

In “[Otros puntos de enganche del lado cliente](#)” we talk briefly about the pre-push hook, which is a script we can setup to run before a push completes to verify that it should be allowed to push.

Finally, in “[Pushing Refspecs](#)” we look at pushing with a full refspec instead of the general shortcuts that are normally used. This can help you be very specific about what work you wish to share.

git remote

The `git remote` command is a management tool for your record of remote repositories. It allows you to save long URLs as short handles, such as “origin” so you don’t have to type them out all the time. You can have several of these and the `git remote` command is used to add, change and delete them.

This command is covered in detail in “[Trabajar con Remotos](#)”, including listing, adding, removing and renaming them.

It is used in nearly every subsequent chapter in the book too, but always in the standard `git remote add <name> <url>` format.

git archive

The `git archive` command is used to create an archive file of a specific snapshot of the project.

We use `git archive` to create a tarball of a project for sharing in “[Preparando una versión](#)”.

git submodule

The `git submodule` command is used to manage external repositories within a normal repositories. This could be for libraries or other types of shared resources. The `submodule` command has several sub-commands (`add`, `update`, `sync`, etc) for managing these resources.

This command is only mentioned and entirely covered in “[Submodules](#)”.

Inspection and Comparison

git show

The `git show` command can show a Git object in a simple and human readable way. Normally you would use this to show the information about a tag or a commit.

We first use it to show annotated tag information in “[Etiquetas Anotadas](#)”.

Later we use it quite a bit in “[Revision Selection](#)” to show the commits that our various revision selections resolve to.

One of the more interesting things we do with `git show` is in “[Manual File Re-merging](#)” to extract specific file contents of various stages during a merge conflict.

git shortlog

The `git shortlog` command is used to summarize the output of `git log`. It will take many of the same options that the `git log` command will but instead of listing out all of the commits it will present a summary of the commits grouped by author.

We showed how to use it to create a nice changelog in “[El registro resumido](#)”.

git describe

The `git describe` command is used to take anything that resolves to a commit and produces a string that is somewhat human-readable and will not change. It’s a way to get a description of a commit that is as unambiguous as a commit SHA-1 but more understandable.

We use `git describe` in “[Generando un número de compilación](#)” and “[Preparando una versión](#)” to get a string to name our release file after.

Debugging

Git has a couple of commands that are used to help debug an issue in your code. This ranges from figuring out where something was introduced to figuring out who introduced it.

git bisect

The `git bisect` tool is an incredibly helpful debugging tool used to find which specific commit was the first one to introduce a bug or problem by doing an automatic binary search.

It is fully covered in “[Binary Search](#)” and is only mentioned in that section.

git blame

The `git blame` command annotates the lines of any file with which commit was the last one to introduce a change to each line of the file and what person authored that commit. This is helpful in order to find the person to ask for more information about a specific section of your code.

It is covered in “[File Annotation](#)” and is only mentioned in that section.

git grep

The `git grep` command can help you find any string or regular expression in any of the files in your source code, even older versions of your project.

It is covered in “**Git Grep**” and is only mentioned in that section.

Patching

A few commands in Git are centered around the concept of thinking of commits in terms of the changes they introduce, as thought the commit series is a series of patches. These commands help you manage your branches in this manner.

git cherry-pick

The `git cherry-pick` command is used to take the change introduced in a single Git commit and try to re-introduce it as a new commit on the branch you’re currently on. This can be useful to only take one or two commits from a branch individually rather than merging in the branch which takes all the changes.

Cherry picking is described and demonstrated in “**Flujos de trabajo reorganizando o entresacando**”.

git rebase

The `git rebase` command is basically an automated `cherry-pick`. It determines a series of commits and then cherry-picks them one by one in the same order somewhere else.

Rebasing is covered in detail in “**Reorganizar el Trabajo Realizado**”, including covering the collaborative issues involved with rebasing branches that are already public.

We use it in practice during an example of splitting your history into two separate repositories in “**Replace**”, using the `--onto` flag as well.

We go through running into a merge conflict during rebasing in “**Rerere**”.

We also use it in an interactive scripting mode with the `-i` option in “**Changing Multiple Commit Messages**”.

git revert

The `git revert` command is essentially a reverse `git cherry-pick`. It creates a new commit that applies the exact opposite of the change introduced in the commit you're targeting, essentially undoing or reverting it.

We use this in “[Reverse the commit](#)” to undo a merge commit.

Email

Many Git projects, including Git itself, are entirely maintained over mailing lists. Git has a number of tools built into it that help make this process easier, from generating patches you can easily email to applying those patches from an email box.

git apply

The `git apply` command applies a patch created with the `git diff` or even `GNU diff` command. It is similar to what the `patch` command might do with a few small differences.

We demonstrate using it and the circumstances in which you might do so in “[Aplicando parches recibidos por e-mail](#)”.

git am

The `git am` command is used to apply patches from an email inbox, specifically one that is `mbox` formatted. This is useful for receiving patches over email and applying them to your project easily.

We covered usage and workflow around `git am` in “[Aplicando un parche con am](#)” including using the `--resolved`, `-i` and `-3` options.

There are also a number of hooks you can use to help with the workflow around `git am` and they are all covered in “[Puntos en el flujo de trabajo del correo electrónico](#)”.

We also use it to apply patch formatted GitHub Pull Request changes in “[Notificaciones por correo electrónico](#)”.

git format-patch

The `git format-patch` command is used to generate a series of patches in `mbox` format that you can use to send to a mailing list properly formatted.

We go through an example of contributing to a project using the `git format-patch` tool in “[Public Project over E-Mail](#)”.

git send-email

The `git send-email` command is used to send patches that are generated with `git format-patch` over email.

We go through an example of contributing to a project by sending patches with the `git send-email` tool in “[Public Project over E-Mail](#)”.

git request-pull

The `git request-pull` command is simply used to generate an example message body to email to someone. If you have a branch on a public server and want to let someone know how to integrate those changes without sending the patches over email, you can run this command and send the output to the person you want to pull the changes in.

We demonstrate how to use `git request-pull` to generate a pull message in “[Forked Public Project](#)”.

External Systems

Git comes with a few commands to integrate with other version control systems.

git svn

The `git svn` command is used to communicate with the Subversion version control system as a client. This means you can use Git to checkout from and commit to a Subversion server.

This command is covered in depth in “[Git and Subversion](#)”.

git fast-import

For other version control systems or importing from nearly any format, you can use `git fast-import` to quickly map the other format to something Git can easily record.

This command is covered in depth in “[A Custom Importer](#)”.

Administration

If you’re administering a Git repository or need to fix something in a big way, Git provides a number of administrative commands to help you out.

git gc

The `git gc` command runs “garbage collection” on your repository, removing unnecessary files in your database and packing up the remaining files into a more efficient format.

This command normally runs in the background for you, though you can manually run it if you wish. We go over some examples of this in “[Maintenance](#)”.

git fsck

The `git fsck` command is used to check the internal database for problems or inconsistencies.

We only quickly use this once in “[Data Recovery](#)” to search for dangling objects.

git reflog

The `git reflog` command goes through a log of where all the heads of your branches have been as you work to find commits you may have lost through rewriting histories.

We cover this command mainly in “[RefLog Shortnames](#)”, where we show normal usage to and how to use `git log -g` to view the same information with `git log` output.

We also go through a practical example of recovering such a lost branch in “[Data Recovery](#)”.

git filter-branch

The `git filter-branch` command is used to rewrite loads of commits according to certain patterns, like removing a file everywhere or filtering the entire repository down to a single subdirectory for extracting a project.

In “[Removing a File from Every Commit](#)” we explain the command and explore several different options such as `--commit-filter`, `--subdirectory-filter` and `--tree-filter`.

In “[Git-p4](#)” and “[TFS](#)” we use it to fix up imported external repositories.

Plumbing Commands

There were also quite a number of lower level plumbing commands that we encountered in the book.

The first one we encounter is `ls-remote` in “[Referencias de Pull Request](#)” which we use to look at the raw references on the server.

We use `ls-files` in “[Manual File Re-merging](#)”, “[Rerere](#)” and “[The Index](#)” to take a more raw look at what your staging area looks like.

We also mention `rev-parse` in “[Branch References](#)” to take just about any string and turn it into an object SHA-1.

However, most of the low level plumbing commands we cover are in [Chapter 10](#), which is more or less what the chapter is focused on. We tried to avoid use of them throughout most of the rest of the book.

Index

Symbols

\$EDITOR, **388**
\$VISUAL
 see \$EDITOR, **388**
.gitignore, **390**
.NET, **556**
{@{upstream}}, **115**
@{u}, **115**
©, **552**

A

aliases, **79**
Apache, **145**
Apple, **556**
archiving, **408**
attributes, **400**
autocorrect, **391**

B

bash, **545**
binary files, **401**
BitKeeper, **30**
bitnami, **149**
branches, **83**
 basic workflow, **91**
 creating, **86**
 diffing, **190**
 long-running, **103**
 managing, **101**
 merging, **96**
 remote, **107, 189**
 switching, **87**
 topic, **185**
 tracking, **114**
 upstream, **114**
build numbers, **199**

C

C#, **556**
Cocoa, **556**
color, **392**
commit templates, **389**
contributing, **161**
 private managed team, **171**
 private small team, **163**
 public large project, **181**
 public small project, **177**
credential caching, **37**
credentials, **379**
CRLF, **37**
crlf, **397**
CVS, **27**

D

difftool, **393**
distributed git, **157**

E

Eclipse, **544**
editor
 changing default, **55**
email, **183**
 applying patches from, **185**
excludes, **390, 492**

F

files
 moving, **58**
 removing, **56**
forking, **159, 209**

G

Git as a client, **427**

git commands
 add, 47, 47, 48
 am, 186
 apply, 185
 archive, 200
 branch, 101
 checkout, 87
 cherry-pick, 196
 clone, 44
 bare, 136
 commit, 54, 84
 config, 39, 40, 55, 79, 183, 387
 credential, 379
 daemon, 143
 describe, 199
 diff, 51
 check, 162
 fast-import, 482
 fetch, 71
 fetch-pack, 518
 filter-branch, 480
 format-patch, 182
 gitk, 536
 gui, 536
 help, 41, 143
 http-backend, 144
 init, 43, 47
 bare, 137, 141
 instaweb, 147
 log, 59
 merge, 94
 squash, 181
 mergetool, 100
 p4, 456, 479
 pull, 72
 push, 72, 78, 112
 rebase, 118
 receive-pack, 516
 remote, 69, 71, 73, 74
 request-pull, 178
 rerere, 197
 send-pack, 516
 shortlog, 201
 show, 77
 show-ref, 430
 status, 46, 54
 svn, 427
 tag, 75, 76, 78
 upload-pack, 518

git-svn, 427
git-tf, 464

git-tfs, 464
GitHub, 203
 API, 252
 Flow, 210
 organizations, 243
 pull requests, 213
 user accounts, 203
GitHub for Mac, 538
GitHub for Windows, 538
gitk, 536
GitLab, 149
GitWeb, 146
GPG, 390
Graphical tools, 535
GUIs, 535

H

hooks, 409
 post-update, 133

I

ignoring files, 50
Importing
 from Mercurial, 476
 from others, 482
 from Perforce, 478
 from Subversion, 474
 from TFS, 481

integrating work, 192

Interoperation with other VCSs
 Mercurial, 439
 Perforce, 448
 Subversion, 427
 TFS, 464

IRC, 41

J

java, 557
jgit, 557

K

keyword expansion, 404

L

libgit2, 552
line endings, 397
Linux, 30

installing, 36
log filtering, 66
log formatting, 62

M

Mac
installing, 36
maintaining a project, 184
master, 85
Mercurial, 439, 476
mergetool, 393
merging, 96
 conflicts, 98
 strategies, 409
 vs. rebasing, 127
Migrating to Git, 473
Mono, 556

O

Objective-C, 556
origin, 107

P

pager, 390
Perforce, 27, 30, 448, 478
 Git Fusion, 448
policy example, 414
posh-git, 548
Powershell, 37
powershell, 548
protocols
 dumb HTTP, 132
 git, 135
 local, 130
 smart HTTP, 132
 SSH, 134
pulling, 116
pushing, 112
Python, 557

R

rebasing, 117
 perils of, 122
 vs. merging, 127
references
 remote, 107
releasing, 200
rerere, 197

Ruby, 553

S

serving repositories, 129
 git protocol, 143
 GitLab, 149
 GitWeb, 146
 HTTP, 144
 SSH, 138
SHA-1, 33
shell prompts
 bash, 545
 powershell, 548
 zsh, 547
SSH keys, 139
 with GitHub, 204
staging area
 skipping, 56
Subversion, 27, 30, 158, 427, 474

T

tab completion
 bash, 545
 powershell, 548
 zsh, 547
tags, 74, 198
 annotated, 76
 lightweight, 76
 signing, 198
TFS, 464, 481
TFVC (see TFS)

V

version control, 25
 centralized, 27
 distributed, 28
 local, 26
Visual Studio, 542

W

whitespace, 396
Windows
 installing, 37
workflows, 157
 centralized, 157
 dictator and lieutenants, 159
 integration manager, 158
 merging, 192

merging (large), **194**
rebasing and cherry-picking, **196**

Z
zsh, **547**

X
Xcode, **36**