

THE EXPERT'S VOICE®

SEGUNDA EDICIÓN

Pro Git

*TODO LO QUE NECESITAS SABER
ACERCA DE GIT*

Scott Chacon y Ben Straub

Apress®

Pro Git

Versión 2.1.22-4-g2264d13, 2021-08-14

Tabla de Contenido

| | | |
|--|-------|-------|
| Licence | | |
| Prefacio por Scott Chacon | | 1 |
| Prefacio por Ben Straub | | 2 |
| Dedicatorias | | 4 |
| Contribuidores | | 5 |
| Introducción | | 6 |
| Inicio - Sobre el Control de Versiones | | 7 |
| Acerca del Control de Versiones | | 9 |
| Una breve historia de Git | | 9 |
| Fundamentos de Git | | 13 |
| La Línea de Comandos | | 13 |
| Instalación de Git | | 17 |
| Configurando Git por primera vez | | 18 |
| ¿Cómo obtener ayuda? | | 20 |
| Resumen | | 22 |
| Fundamentos de Git | | 23 |
| Obteniendo un repositorio Git | | 24 |
| Guardando cambios en el Repositorio | | 24 |
| Ver el Historial de Confirmaciones | | 25 |
| Deshacer Cosas | | 38 |
| Trabajar con Remotos | | 45 |
| Etiquetado | | 48 |
| Alias de Git | | 53 |
| Resumen | | 57 |
| Ramificaciones en Git | | 58 |
| ¿Qué es una rama? | | 59 |
| Procedimientos Básicos para Ramificar y Fusionar | | 60 |
| Gestión de Ramas | | |
| Flujos de Trabajo Ramificados | | 75 |
| Ramas Remotas | | 76 |
| Reorganizar el Trabajo Realizado | | 80 |
| Recapitulación | | 90 |
| Git en el Servidor | | 99 |
| Los Protocolos | | 100 |
| Configurando Git en un servidor | | 100 |
| Generando tu clave pública SSH | | 105 |
| Configurando el servidor | | 108 |
| El demonio Git | | 109 |

| | |
|---|-----|
| HTTP Inteligente | 112 |
| GitWeb | 113 |
| GitLab | 115 |
| Git en un alojamiento externo | 117 |
| Resumen | 121 |
| Git en entornos distribuidos | 122 |
| Flujos de trabajo distribuidos | 123 |
| Contribuyendo a un Proyecto | 123 |
| Manteniendo un proyecto | 127 |
| Resumen | 149 |
| GitHub | 164 |
| Creación y configuración de la cuenta | 165 |
| Participando en Proyectos | 165 |
| Mantenimiento de un proyecto | 171 |
| Gestión de una organización | 190 |
| Scripting en GitHub | 205 |
| Resumen | 208 |
| Herramientas de Git | 219 |
| Revisión por selección | 221 |
| Organización interactiva | 221 |
| Guardado rápido y Limpieza | 229 |
| Firmando tu trabajo | 234 |
| Buscando | 240 |
| Reescribiendo la Historia | 244 |
| Reiniciar Desmitificado | 248 |
| Fusión Avanzada | 255 |
| Rerere | 277 |
| Haciendo debug con Git | 296 |
| Submódulos | 303 |
| Agrupaciones | 306 |
| Replace | 326 |
| Almacenamiento de credenciales | 330 |
| Resumen | 338 |
| Personalización de Git | 344 |
| Configuración de Git | 345 |
| Git Attributes | 345 |
| Puntos de enganche en Git | 356 |
| Un ejemplo de implantación de una determinada política en Git | 369 |
| Recapitulación | 379 |
| Git y Otros Sistemas | 381 |

| | |
|---|-----|
| Migración a Git | 381 |
| Resumen | 428 |
| Los entresijos internos de Git | 444 |
| Fontanería y porcelana | 445 |
| Los objetos Git | 445 |
| Referencias Git | 446 |
| Archivos empaquetadores | 457 |
| Las especificaciones para hacer referencia a... (refspec) | 463 |
| Protocolos de transferencia | |
| Mantenimiento y recuperación de datos | 467 |
| Variables de entorno | 473 |
| Recapitulación | 481 |
| Git en otros entornos | 487 |
| Interfaces gráficas | 488 |
| Git en Visual Studio | 488 |
| Git en Eclipse | 494 |
| Git con Bash | 495 |
| Git en Zsh | 496 |
| Git en Powershell | 497 |
| Resumen | 499 |
| Apéndice A: Integrando Git en tus Aplicaciones | 500 |
| Git mediante Línea de Comandos | |
| Libgit2 | 501 |
| JGit | 501 |
| Apéndice B: Comandos de Git | 507 |
| Configuración | 511 |
| Obtener y Crear Proyectos | 511 |
| Seguimiento Básico | 512 |
| Ramificar y Fusionar | 513 |
| Compartir y Actualizar Proyectos | 516 |
| Inspección y Comparación | 518 |
| Depuración | 521 |
| Parcheo | 521 |
| Correo Electrónico | 522 |
| Sistemas Externos | 523 |
| Administración | 524 |
| Comandos de Fontanería | 524 |
| | 525 |

Licence

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 puse 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 desagrada 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 decides 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.

| | | |
|---------------------------------|-----------------------------|---------------------|
| Aleh Suprunovich | José Antonio Muñoz Jiménez | Sanders Kleinfeld |
| Alexandre Garnier | José Carlos García | Santiago Aramis |
| Andres Mancera | Juan | Sarah Schneider |
| Andrew MacFie | Juan Miguel Jimenez | Siarhei Krukau |
| Antonino Ingargiola | Juan Pablo | Stephan van Maris |
| Arnaud Roig Ninerola | Juan Pablo Flores | Steven Roddis |
| Carlos A. Henri quez Q | Juan Sebastián Casallas | Thomas Ackermann |
| Carlos Martín Nieto | Juane99 | Tom Schady |
| Changwoo Park | Juanjo Amor | Vladimir Rodríguez |
| Christoph Prokop | Justin Clift | Xxdaniels751xX |
| Christopher Díaz | Louise Corrigan | YoandyShyno |
| Christopher Díaz Riveros | Luc Morin | amabelster |
| Christopher Wilson | Manuel | andres-mancera |
| Cristos A. Ruiz | Mario Rincon | blasillo |
| Damien Tournoud | Marti Bolivar | devwebcl |
| Dan Schmidt | Masood Fallahpoor | dualsky |
| David Bucci | Matthew Miner | herrmartell |
| David Munoz | Michael MacAskill | josue33 |
| DiegoFRamirez | Michael Welch | juliojgd |
| Dmitri Tikhonov | Mike Charles | leoelz |
| Eliecer Daza | Mike Thibodeau | michaelizer |
| Enrique Matías Sánchez (Quique) | Moises Ariel Hernández Rojo | moisesroj0 |
| Fernando Guerra | Nils Reuße | paveljanik |
| GWC | Pablo Schläpfer | petsuter |
| Gabriel O. Mendivil | Pascal Borreli | pilarArr |
| German Gonzalez | Pavel Janík | rahrah |
| Gytree | Philippe Miossec | salvadormf |
| Haruo Nakayama | Pilar Arr | sanders@oreilly.com |
| Jean-Noël Avila | Rayo VM | xJom |
| Joaquin F. Herranz | Roberto | yesmin41 |
| Jon Forrest | Ronald Wampler | |
| Jon Freed | Samuel Castillo | |

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 ti. 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 (VCSSs, 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 VCSSs, 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 formará 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 ti.

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 contribuciones 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 ti. Si estás 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.

Inicio - Sobre el Control de Versiones

Este capítulo se va a hablar de cómo comenzar a utilizar Git. Empezaremos describiendo algunos conceptos básicos sobre las herramientas de control de versiones; después, trataremos de explicar 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 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.

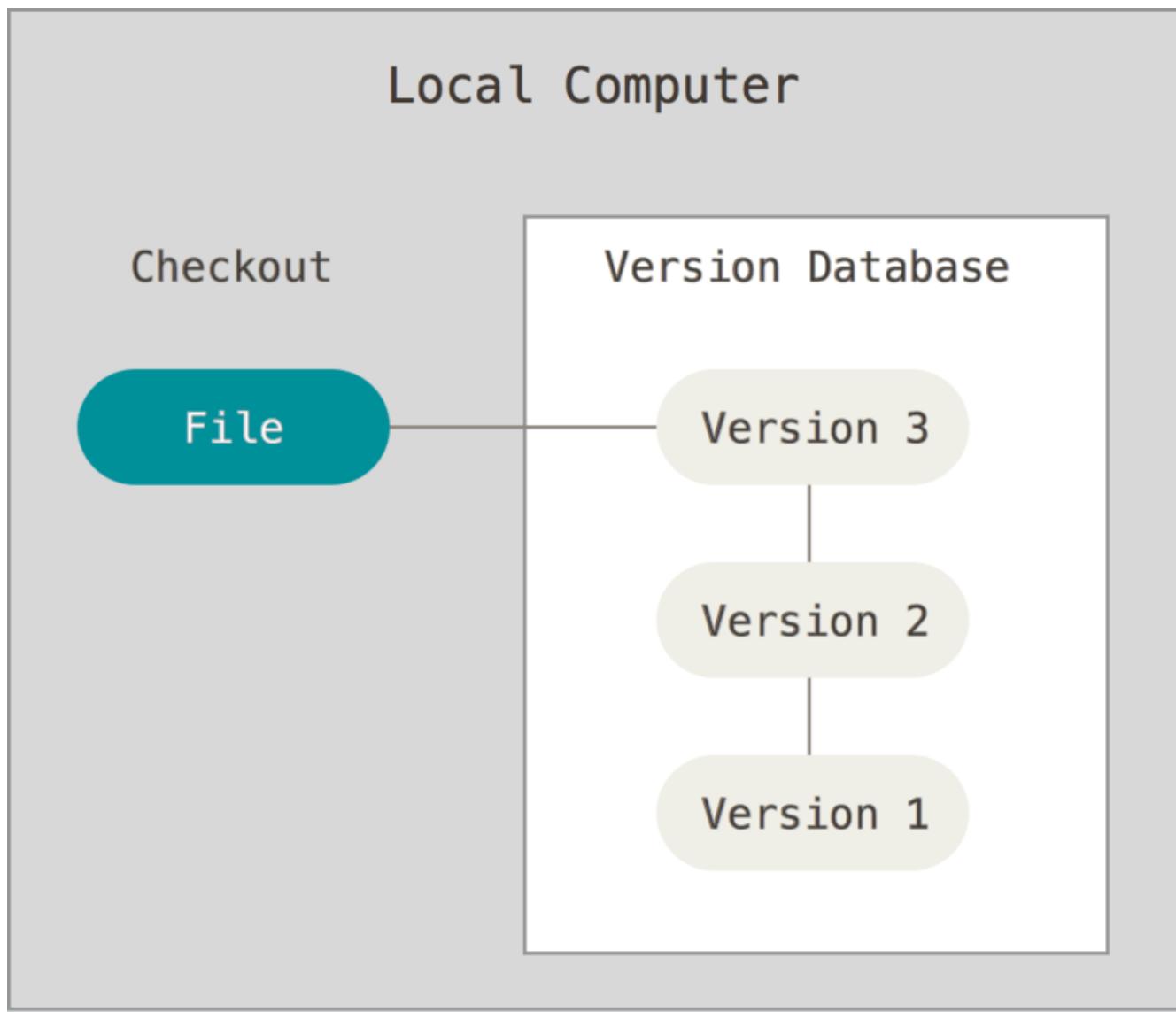


Figura 1. control de versiones local.

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.

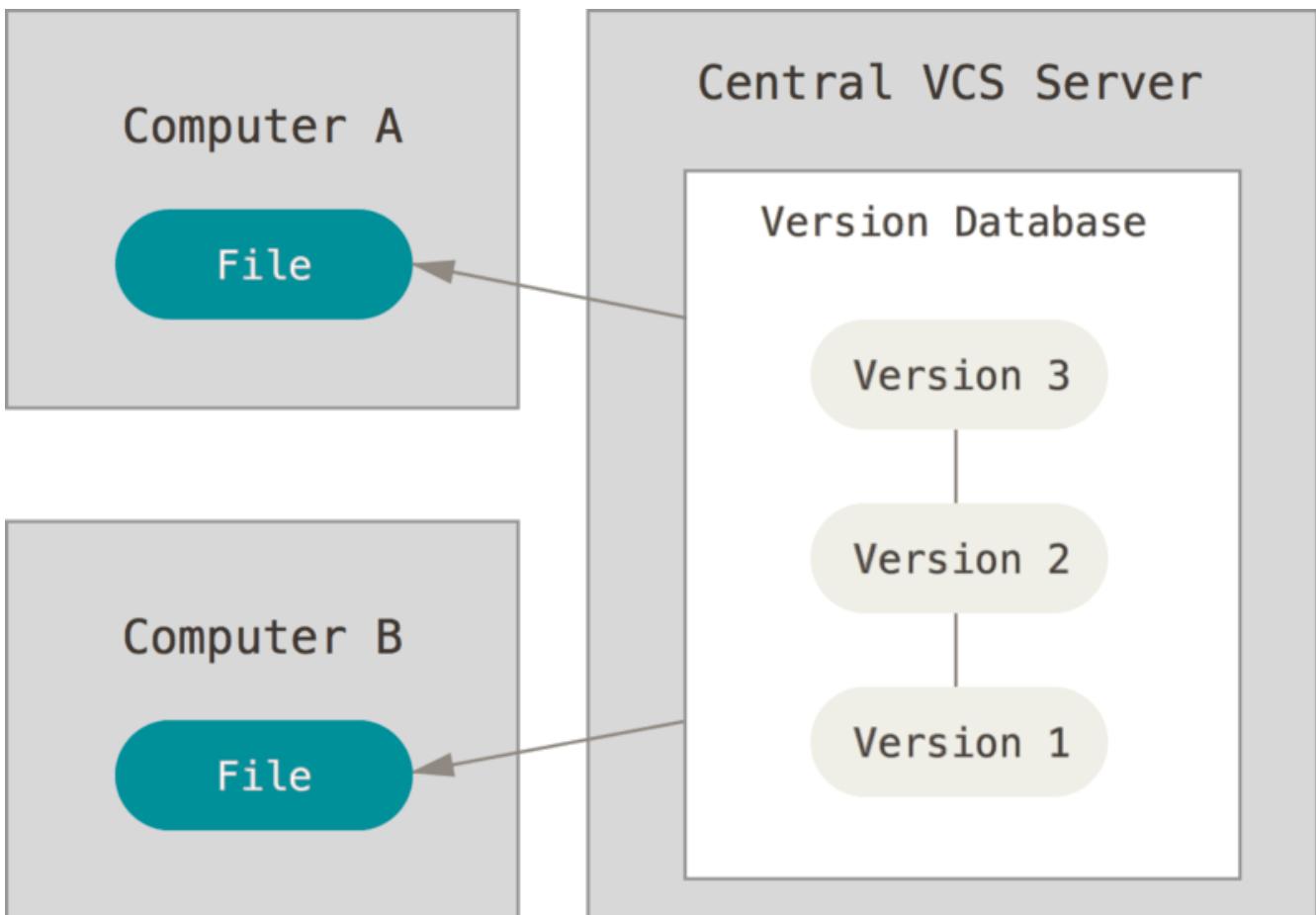


Figura 2. Control de versiones centralizado.

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.

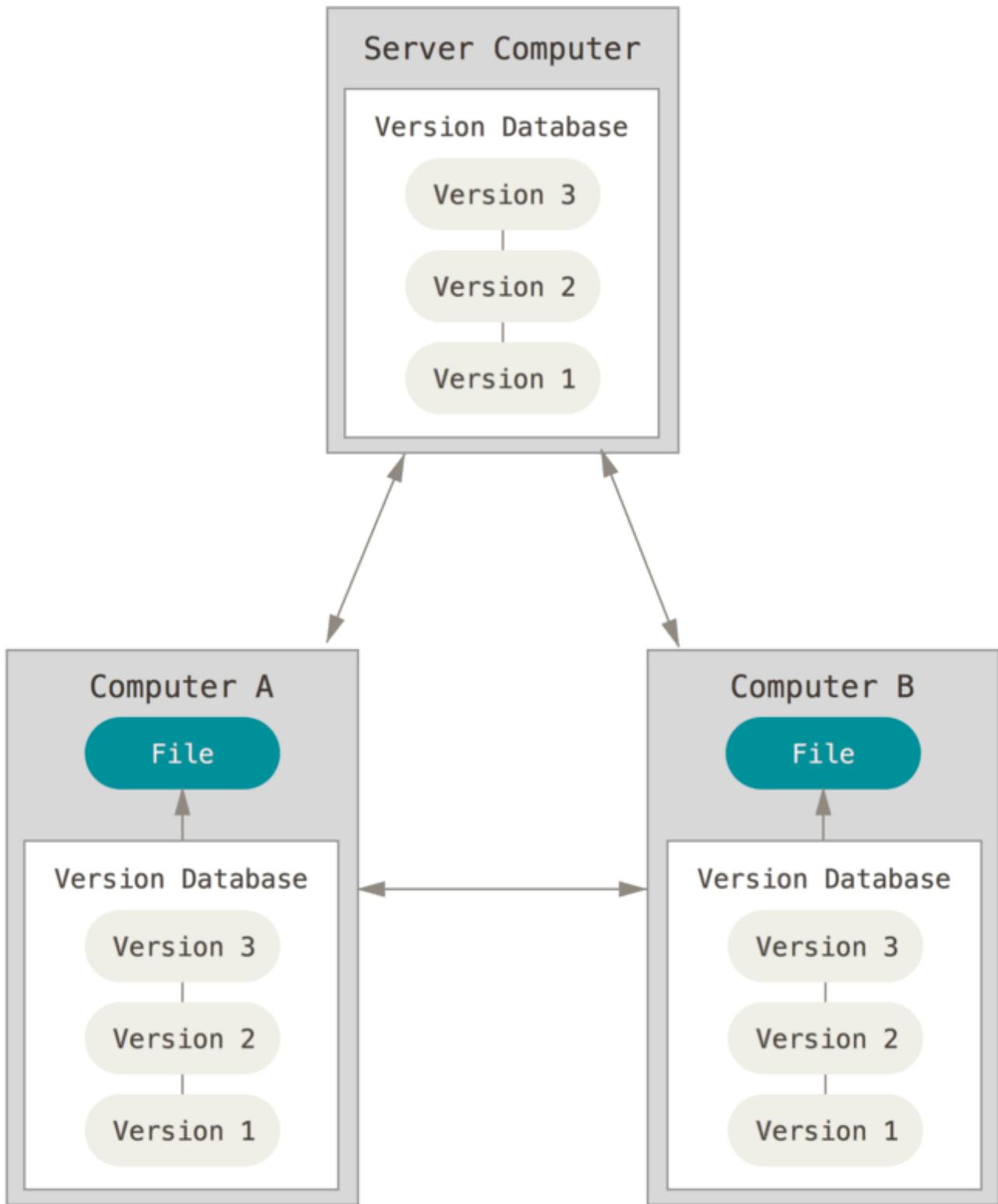


Figura 3. Control de versiones distribuido.

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 [Ramificaciones en Git](#))

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

almacenar como un conjunto de archivos y las modificaciones hechas a cada uno de ellos a través del tiempo.

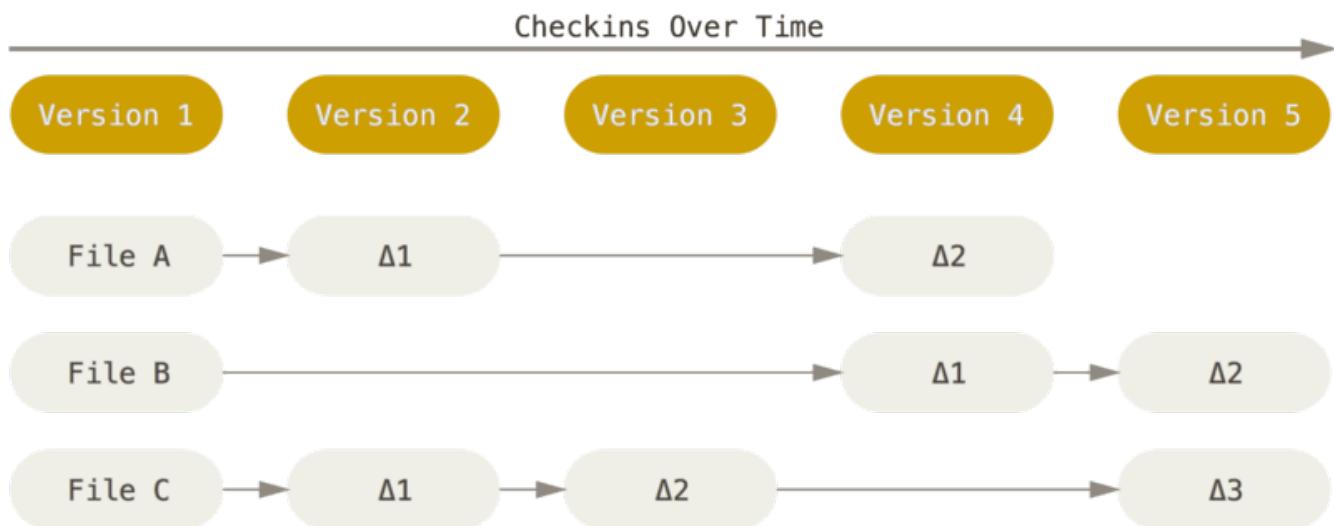


Figura 4. Almacenamiento de datos como cambios en una versión de la base de cada archivo.

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.

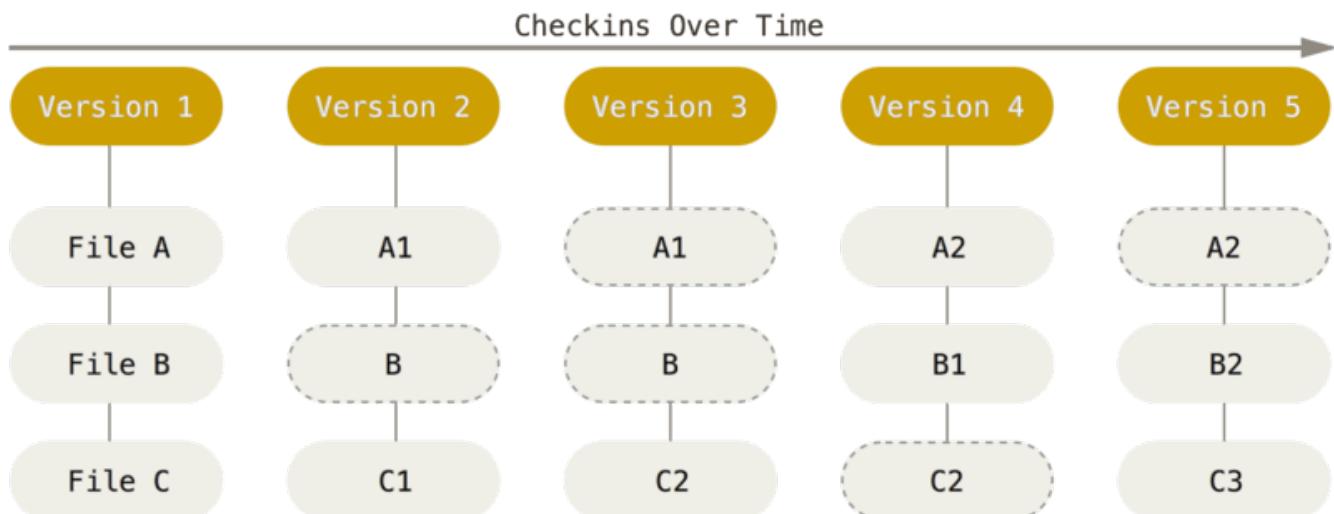


Figura 5. Almacenamiento de datos como instantáneas del proyecto a través del tiempo.

Esta es una diferencia importante entre Git y prácticamente todos los demás VCS. Hace que Git reconsidera 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 [Ramificaciones en Git](#)).

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 computador 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 quieras ver los cambios introducidos en un archivo entre la versión actual y la de hace un mes, Git puede buscar el archivo de 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 con base en 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 sólo 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 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](#).

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).

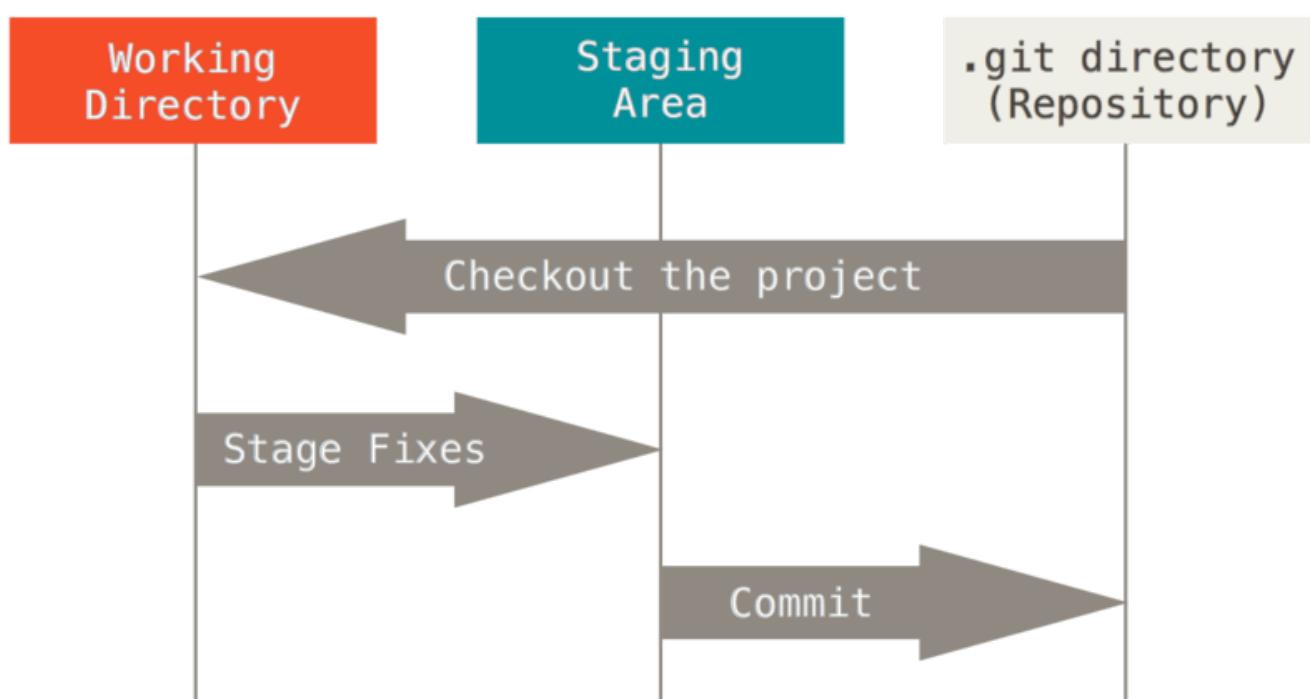


Figura 6. Directorio de trabajo, área de almacenamiento y el directorio Git.

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 [Fundamentos de Git](#) 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 este 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.

NOTA

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 lo hagan ligeramente diferente si estás utilizando una versión anterior de Git. Debido a que Git es particularmente 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 de 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 a 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>.

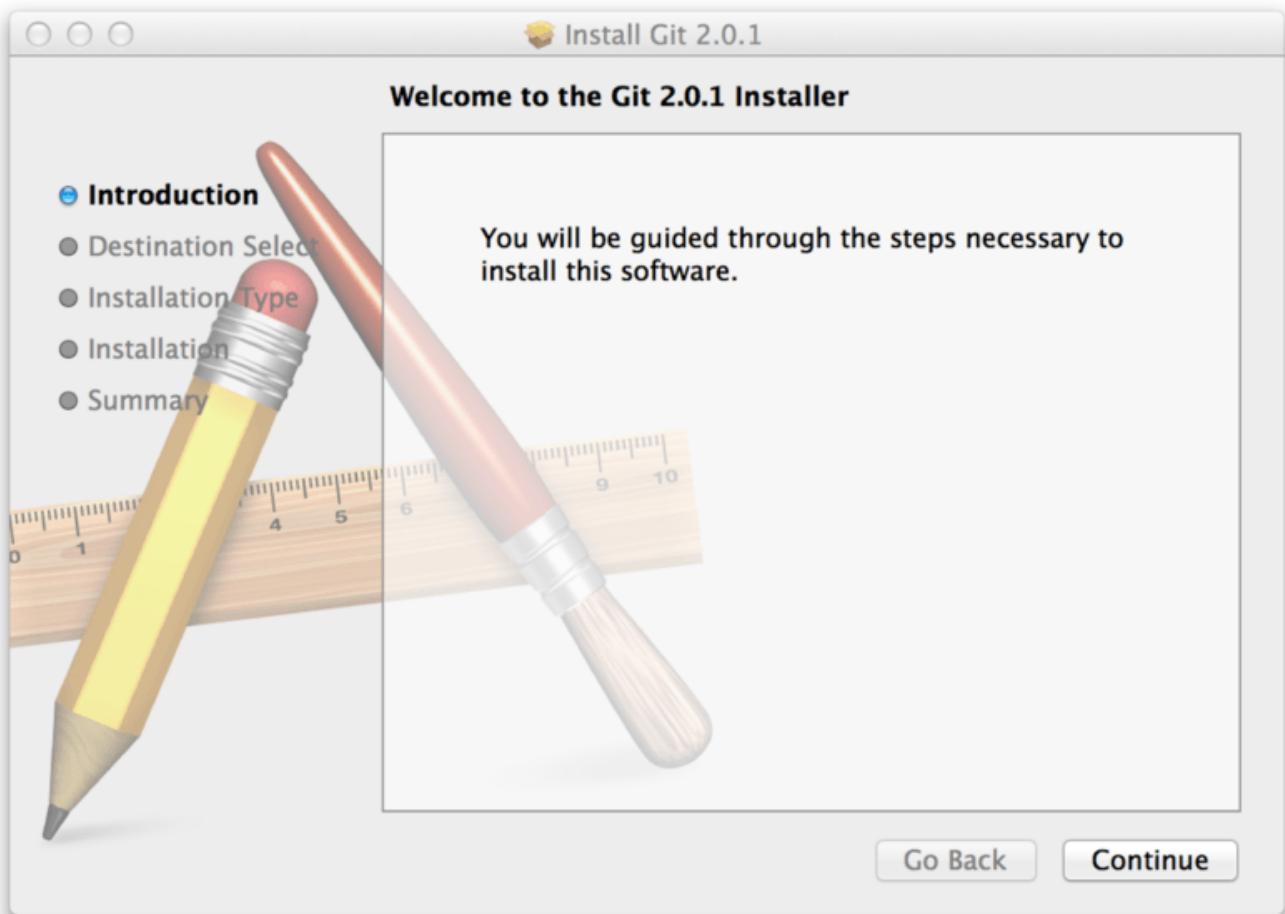


Figura 7. Instalador de Git en OS X.

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. Aprenderemos 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 obtendrán una versión más reciente. Los instaladores binarios tienden 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 obtenerla 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 de 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 del 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 instales 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, sólo 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 usará 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
```

NOTA

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 quieras 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 estos casos, Git usa el último valor para cada clave única que ve.

También puedes comprobar el 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](irc://irc.freenode.net)). Estos canales están llenos de cientos de personas que conocen muy bien Git y suelen estar dispuestos a ayudar.

Resumen

En este momento debes tener una comprensión básica de lo que es Git, y en qué 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

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 [Los entresijos internos de Git](#) 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 el 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 está 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.

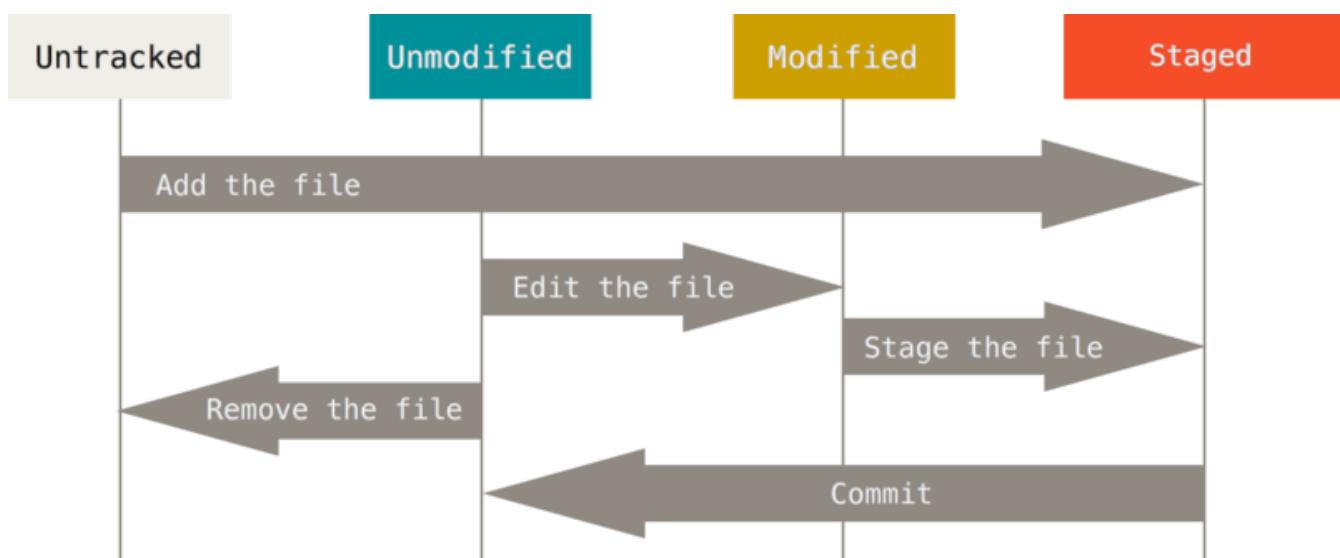


Figura 8. El ciclo de vida del estado de tus archivos.

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 archivos 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 de momento. [Ramificaciones en Git](#) 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 aún 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 archivos en conflicto por combinación como resueltos. Es más útil que lo veas como un comando para “añadir este contenido a la próxima confirmación” más 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 archivo, 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 status`. 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
```

Estado 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 estado 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 compilación. 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 (`~`), la cual es usada 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 comienzan con `#`.
- Emplear patrones glob estándar que se aplicarán recursivamente a todo el directorio del repositorio local.
- Los patrones pueden comenzar en barra (`/`) para evitar recursividad.
- 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 cualquiera; 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 había 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 no este: doc/server/arch.txt
doc/*.txt

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

SUGERENCIA

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 probablemente 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 you 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.

Si quieres 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 ya 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/PROJECTS.md).
+## 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 you 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

NOTA

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 listo 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 [Inicio - Sobre el Control de Versiones](#)).

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 quieras, el área de preparación es a veces un paso más complejo de 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 archivos 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 estado, 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
```

Opciones útiles de `git log --pretty=format` lista algunas de las opciones más útiles aceptadas por `format`.

Tabla 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 |
| <code>%an</code> | Nombre del autor |
| <code>%ae</code> | Dirección de correo del autor |
| <code>%ad</code> | Fecha de autoría (el formato respeta la opción <code>--date</code>) |
| <code>%ar</code> | Fecha de autoría, relativa |
| <code>%cn</code> | Nombre del confirmador |
| <code>%ce</code> | Dirección de correo del confirmador |
| <code>%cd</code> | Fecha de confirmación |
| <code>%cr</code> | Fecha de confirmación, relativa |
| <code>%s</code> | 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 con mayor profundidad en [Git en entornos distribuidos](#).

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. [Opciones típicas de `git log`](#) lista las opciones vistas hasta ahora, y algunas otras opciones de formateo que pueden resultarte útiles, así como su efecto sobre la salida.

Tabla 2. Opciones típicas de `git log`

| Opción | Descripción |
|------------------------------|--|
| <code>-p</code> | Muestra el parche introducido en cada confirmación. |
| <code>--stat</code> | Muestra estadísticas sobre los archivos modificados en cada confirmación. |
| <code>--shortstat</code> | Muestra solamente la línea de resumen de la opción <code>--stat</code> . |
| <code>--name-only</code> | Muestra la lista de archivos afectados. |
| <code>--name-status</code> | Muestra la lista de archivos afectados, indicando además si fueron añadidos, modificados o eliminados. |
| <code>--abbrev-commit</code> | Muestra solamente los primeros caracteres de la suma SHA-1, en vez de los 40 caracteres de que se compone. |
| <code>--relative-date</code> | Muestra la fecha en formato relativo (por ejemplo, “2 weeks ago” (“hace 2 semanas”)) en lugar del formato completo. |
| <code>--graph</code> | Muestra un gráfico ASCII con la historia de ramificaciones y uniones. |
| <code>--pretty</code> | Muestra las confirmaciones usando un formato alternativo. Posibles opciones son <code>oneline</code> , <code>short</code> , <code>full</code> , <code>fuller</code> y <code>format</code> (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 pagina 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 [Opciones para limitar la salida de git log](#) se listan estas opciones, y algunas otras bastante comunes a modo de referencia.

Tabla 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 sólo aquellas confirmaciones cuyo autor coincide con la cadena especificada. |
| <code>--committer</code> | Muestra sólo aquellas confirmaciones cuyo confirmador coincide con la cadena especificada. |

| Opción | Descripción |
|--------|---|
| -S | Muestra sólo aquellas confirmaciones que añaden 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
branch
```

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 quieres 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.

NOTA `git reset` puede ser un comando peligroso, especialmente si lo llamas con la opción `--hard`. Sin embargo, en el escenario descrito anteriormente, el archivo que está en tu directorio de trabajo no se toca, por lo que es relativamente seguro.

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 cómo dominarlo para que haga cosas realmente interesantes en [Reiniciar Desmitificado](#).

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.

IMPORTANTE

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 quieres 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 [Ramificaciones en Git](#); 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 [Recuperación de datos](#) 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 o 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 enviando y trayendo 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, 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 aún 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 [Ramificaciones en Git](#).)

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 [Ramificaciones en Git](#)), 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 [Ramificaciones en Git](#).

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 con tu rama local, 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      (up to
date)
    markdown-strip pushes to markdown-strip  (up to
date)
    master         pushes to master        (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 aún, 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 ha 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 sólo 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 hacerlo es especificar la opción `-a` cuando ejecutas el comando `git 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 <schacon@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'
0b7434d86859cc7b8c3d5e1dddfed66ff742fcbe added a commit function
4682c3261057305bdd616e23b64b0857d832627b added a todo file
166ae0c4d3f420721acbb115cc33848dfcc2121a started write support
9fcebe02d0ae598e95dc970b74767f19372d61af8 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 9fcebe02
```

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 9fce802d0ae598e95dc970b74767f19372d61af8
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 - puedes ejecutar `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 para ellos.

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.

Ramificaciones en Git

Cualquier sistema de control de versiones moderno tiene algún mecanismo para soportar el uso de ramas. 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 muchos 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 [Inicio - Sobre el Control de Versiones](#), 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 confirmaciones (commit) que sean padres directos de esta (un padre 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 [Inicio - Sobre el Control de Versiones](#)), 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.

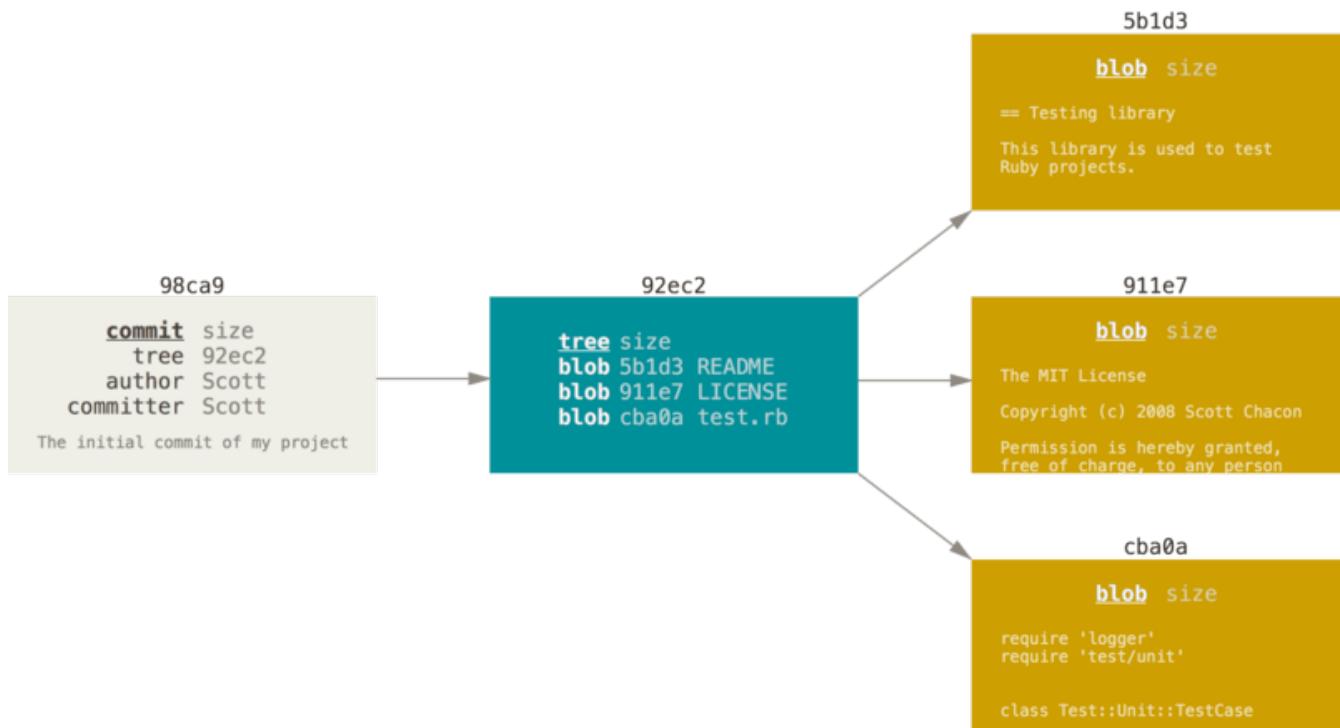


Figura 9. Una confirmación y sus árboles

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

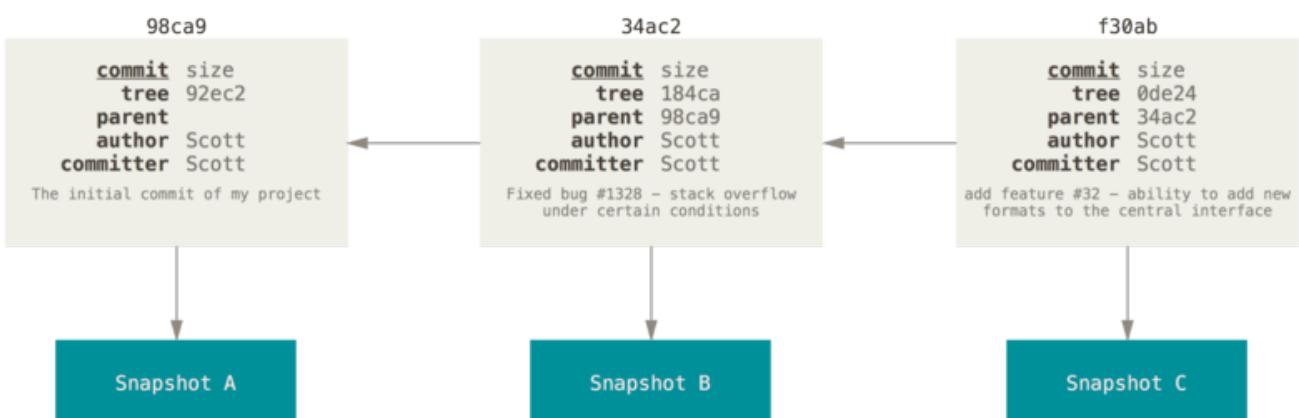


Figura 10. Confirmaciones y sus predecesoras

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.

NOTA

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.

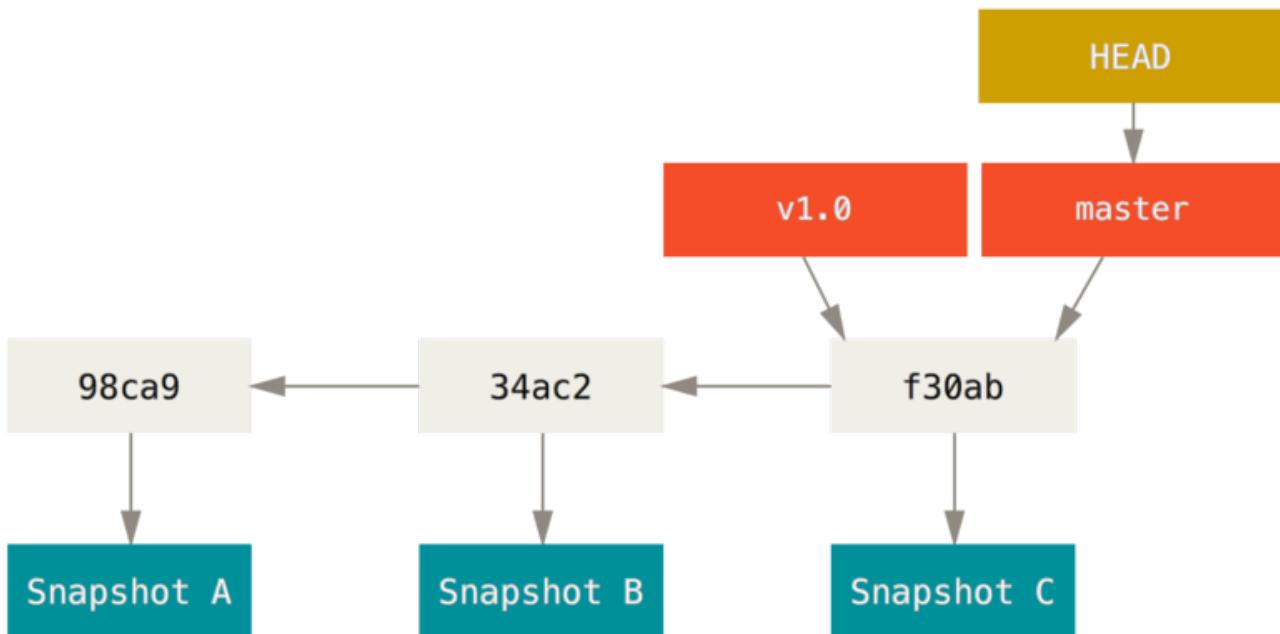


Figura 11. Una rama y su historial de confirmaciones

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.

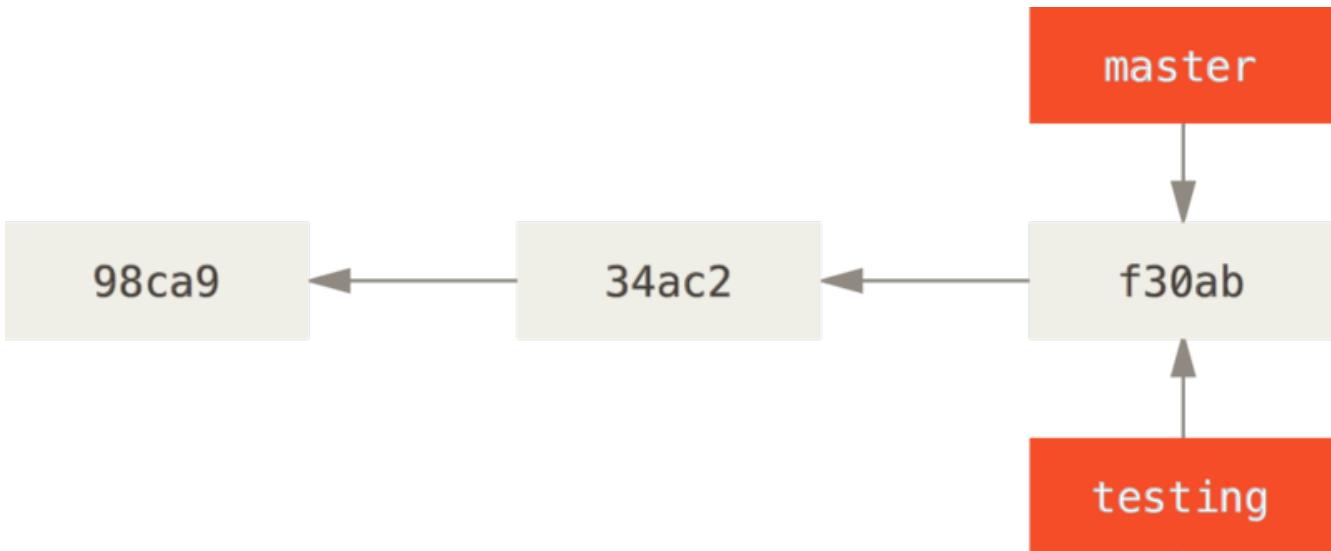


Figura 12. Dos ramas apuntando al mismo grupo de confirmaciones

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, pero no salta a dicha rama.

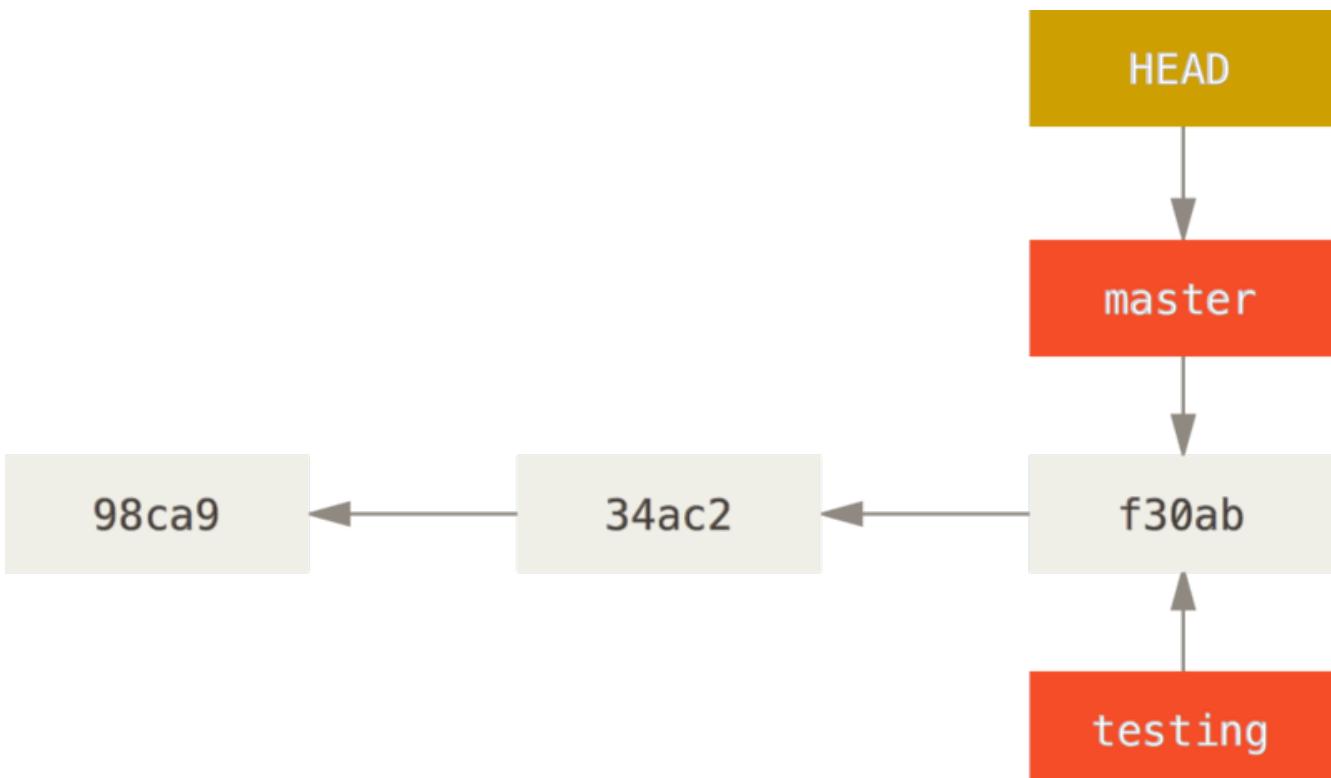


Figura 13. Apuntador HEAD a la rama donde estás actualmente

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**.

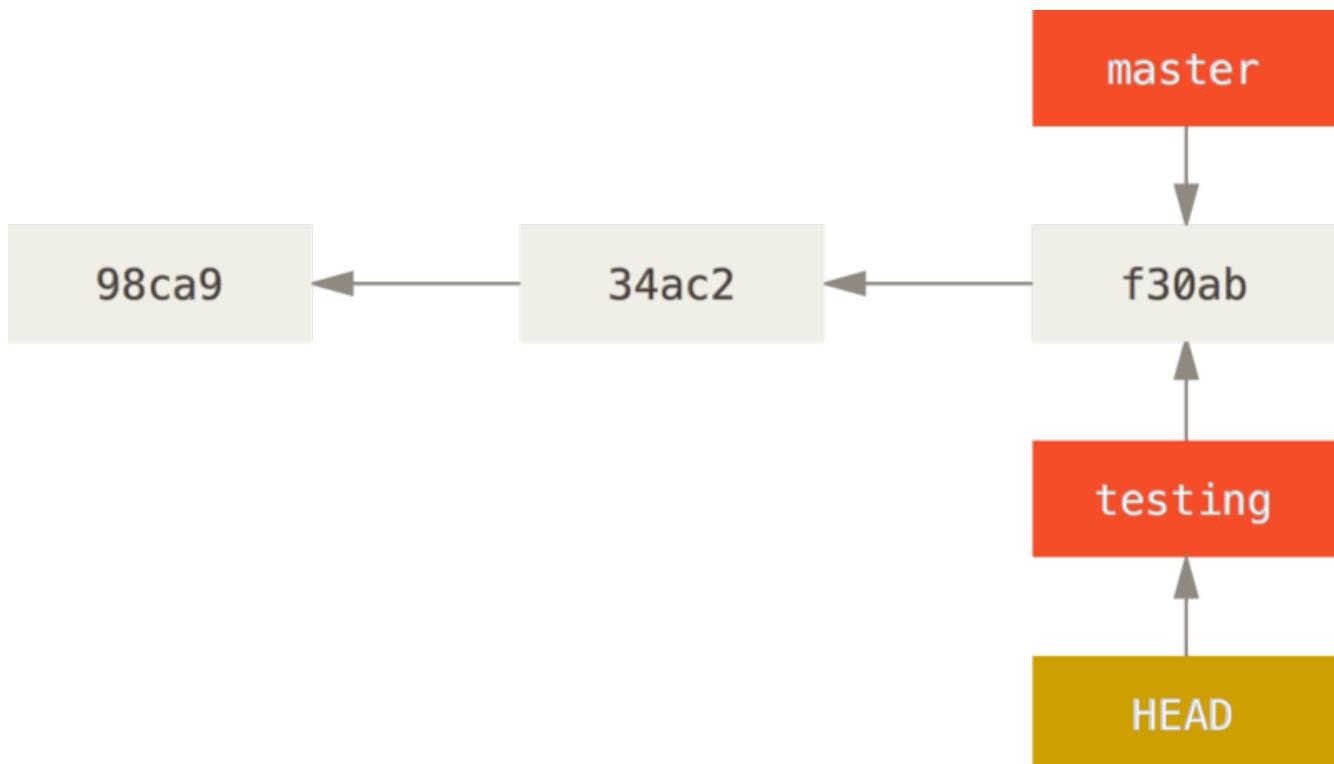


Figura 14. El apuntador HEAD apunta a la rama actual

¿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'
```

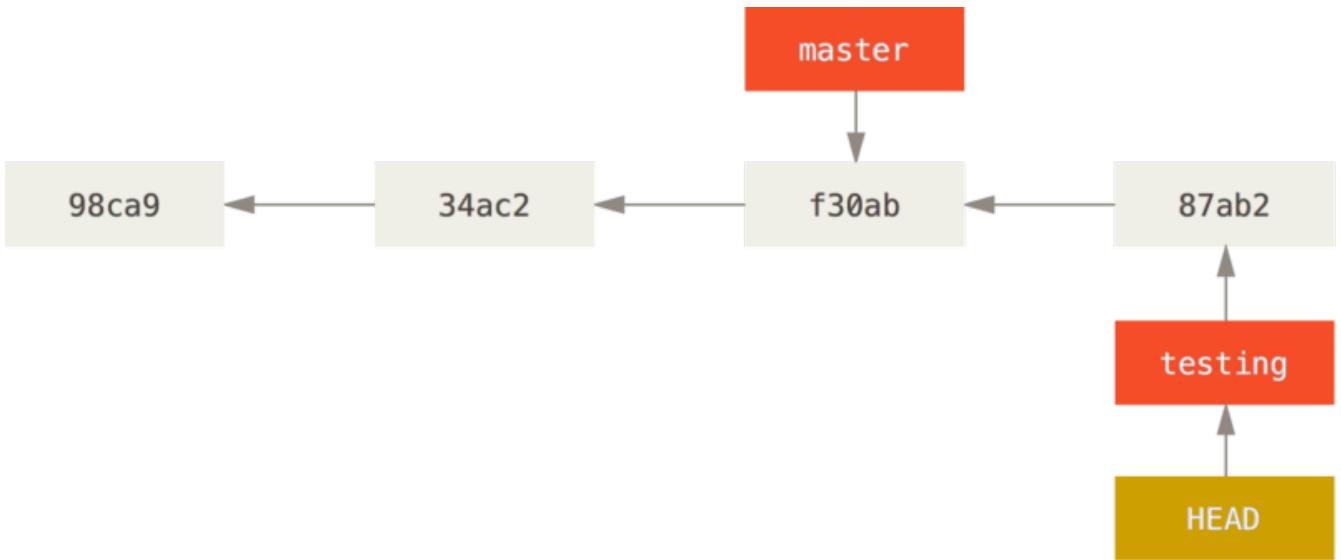


Figura 15. La rama apuntada por HEAD avanza con cada confirmación de cambios

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
```

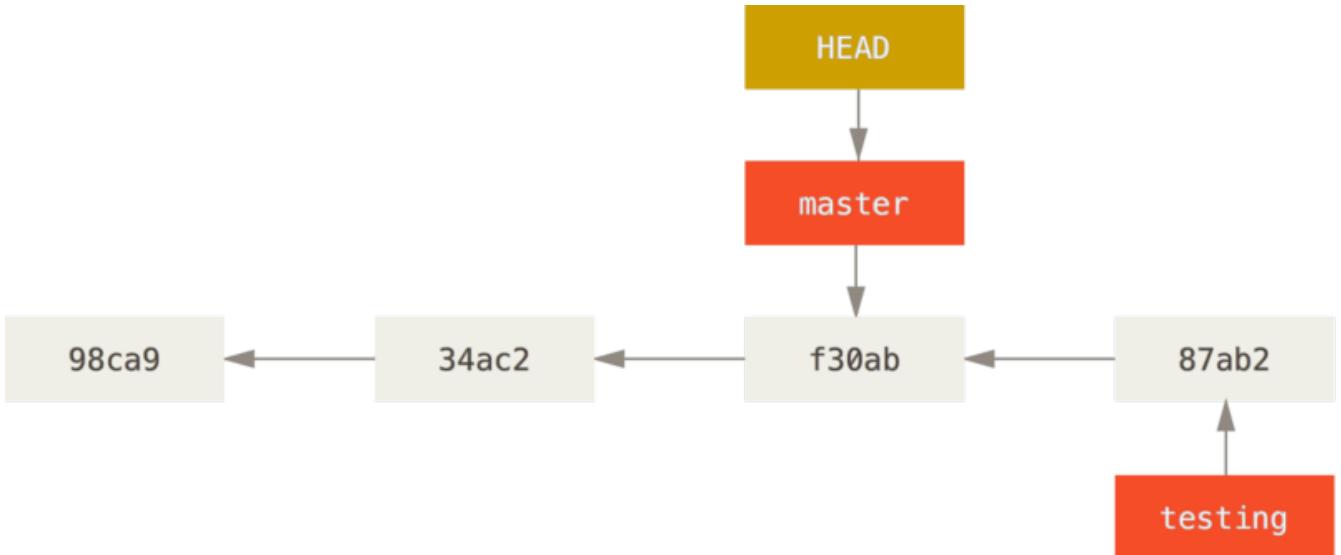


Figura 16. HEAD apunta a otra rama cuando hacemos un salto

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.

NOTA*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 hacía 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 confímalos:

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

Ahora el historial de tu proyecto diverge (ver [Los registros de las ramas divergen](#)). 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`.

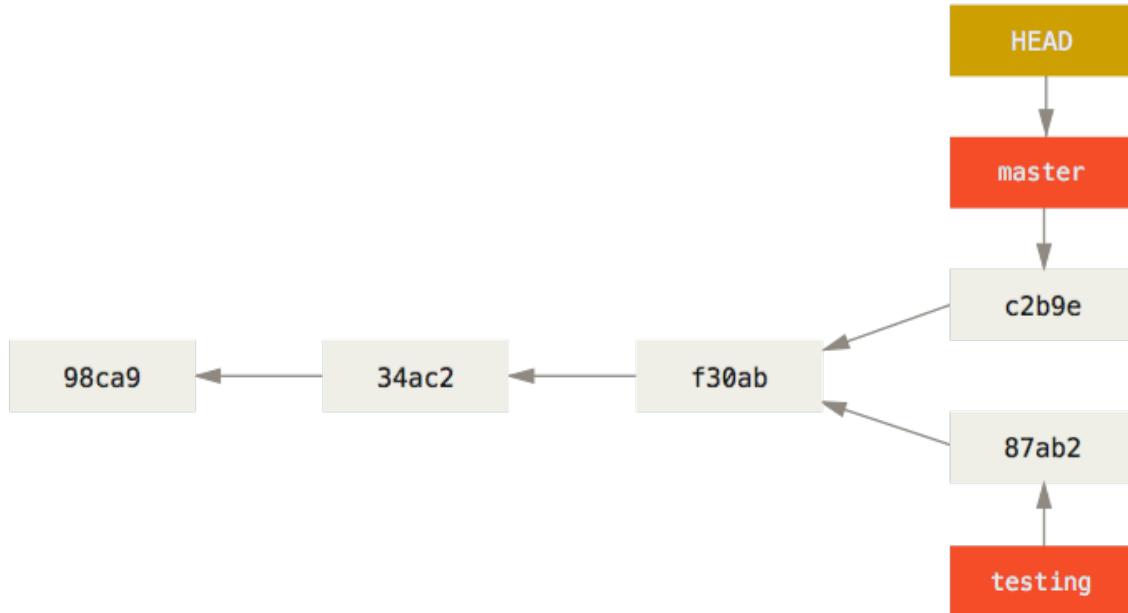


Figura 17. Los registros de las ramas divergen

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 historial.

```
$ 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 supone 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 estas trabajando en un proyecto y tienes un par de confirmaciones (commit) ya realizadas.

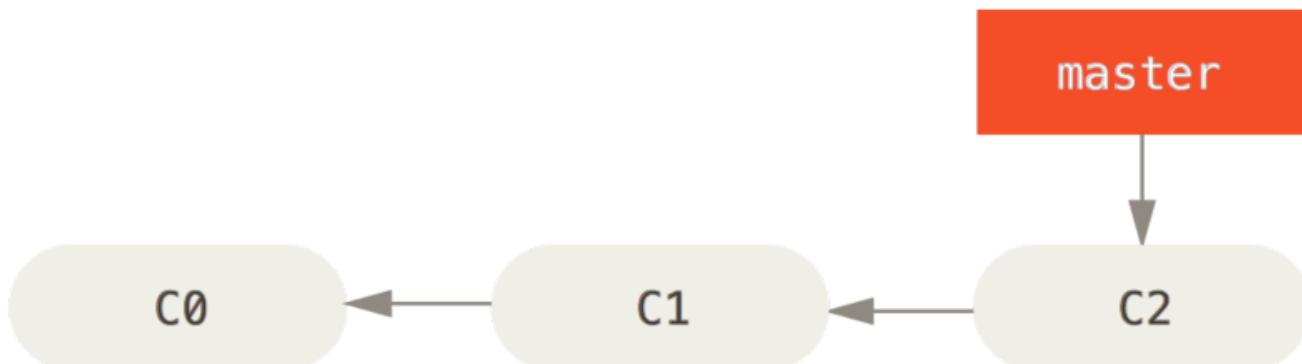


Figura 18. Un registro de confirmaciones corto y sencillo

Decides trabajar en el problema #53, según el sistema que tu compañía utiliza para llevar el 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 para:

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

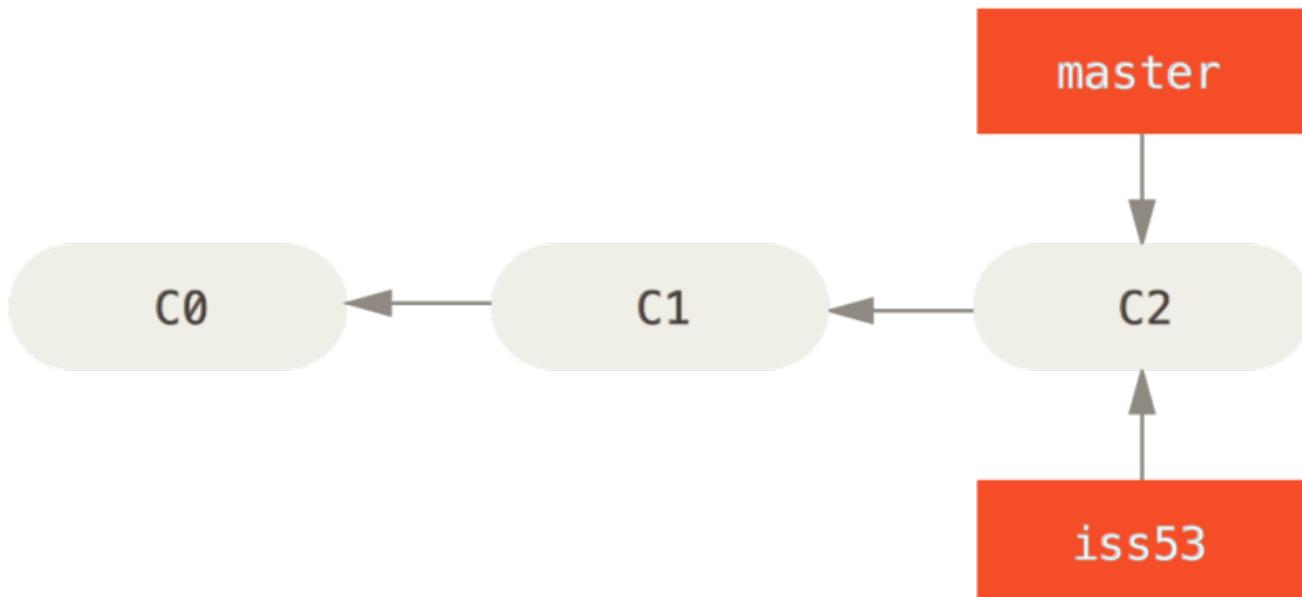


Figura 19. Crear un apuntador a la rama nueva

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]'
```

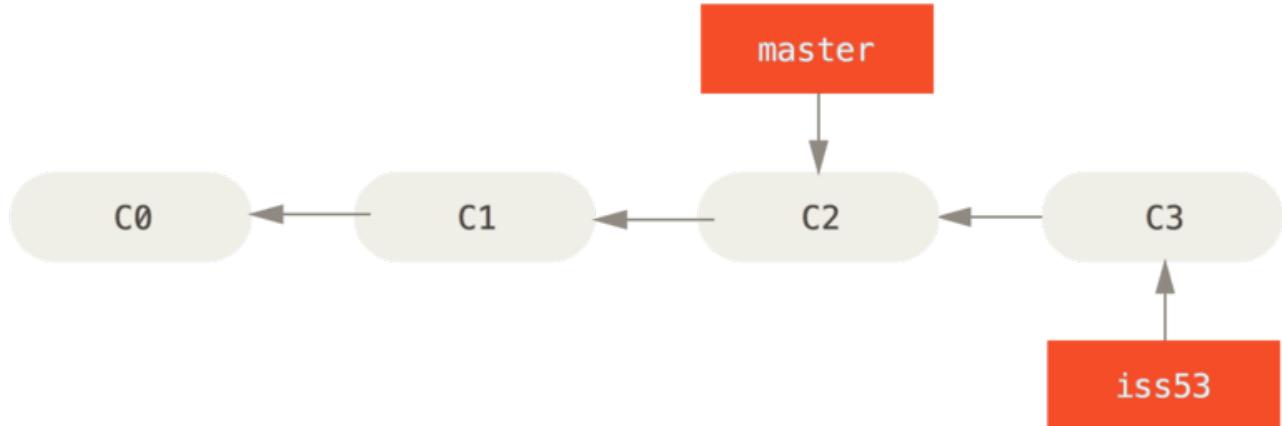


Figura 20. La rama `iss53` ha avanzado con tu trabajo

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 tomar 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 [Guardado rápido y Limpieza](#). 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(+)
```

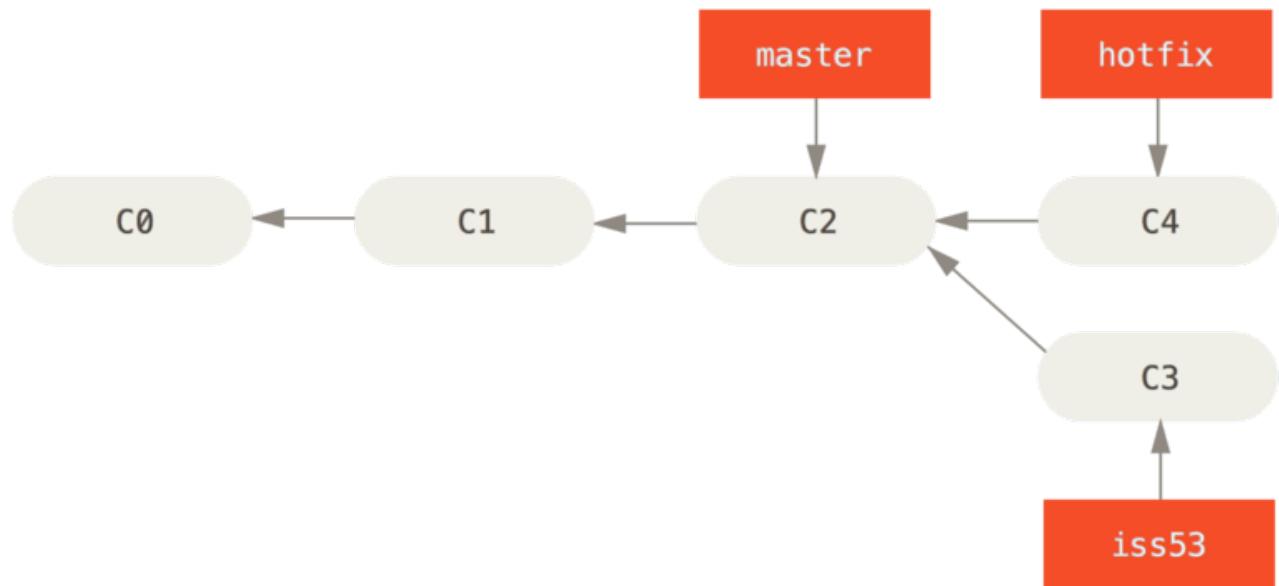


Figura 21. Rama **hotfix** basada en la rama **master** original

Puedes realizar las pruebas oportunas, asegurarte de 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.

Figura 22. 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(+)
```

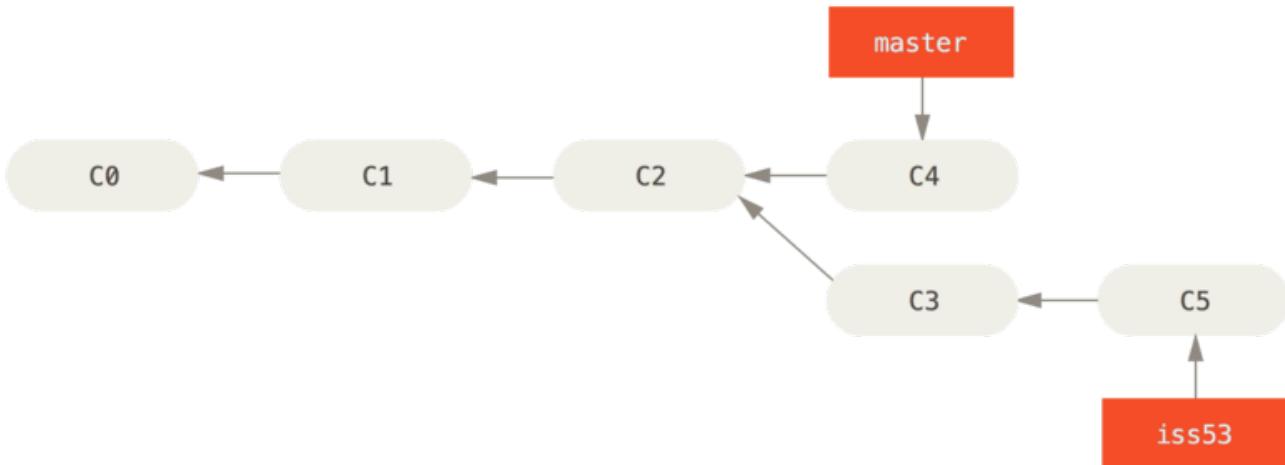


Figura 23. La rama `iss53` puede avanzar independientemente

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.

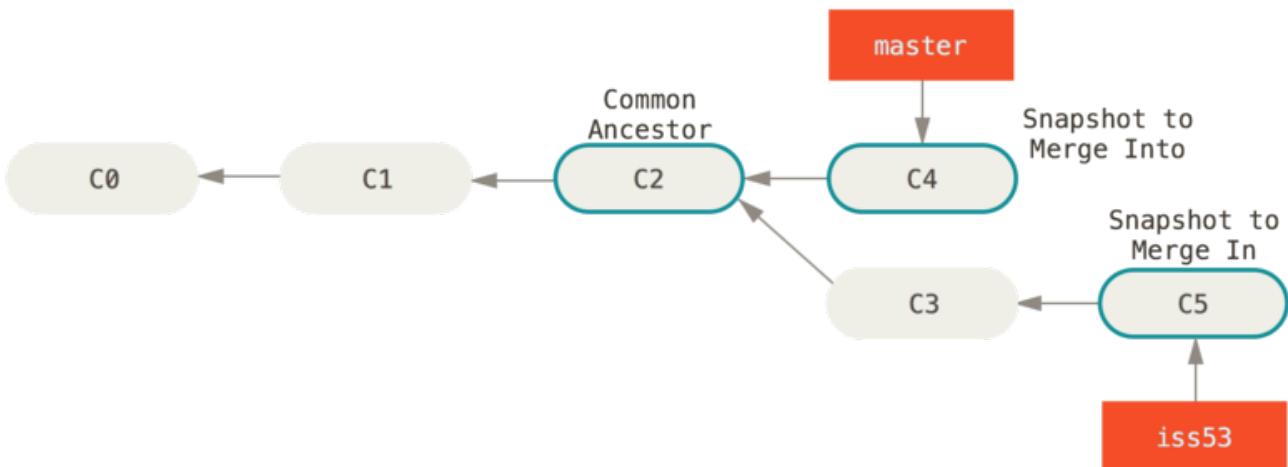


Figura 24. Git identifica automáticamente el mejor ancestro común para realizar la fusión de las ramas

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.

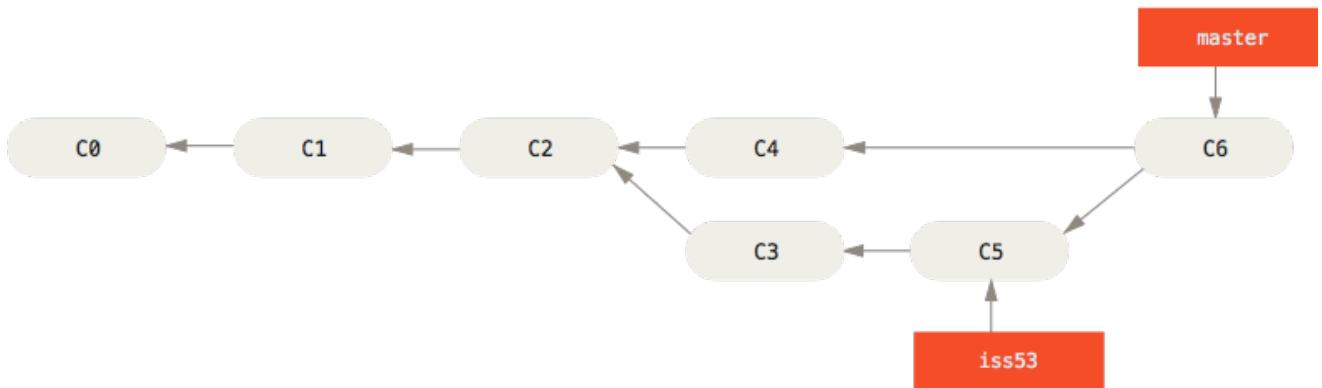


Figura 25. Git crea automáticamente una nueva confirmación para la fusión

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 conflicto. 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 archivo 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
p4merge araxis bc3 codecompare vimdiff emerge
Merging:
index.html

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

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.

NOTA

Si necesitas herramientas más avanzadas para resolver conflictos de fusión más complicados, revisa la sección de fusionado en [Fusión Avanzada](#).

Tras salir de la herramienta de fusionado, 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 con 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 intentar 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, e ir las 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 suelen 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 apuntadores 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.

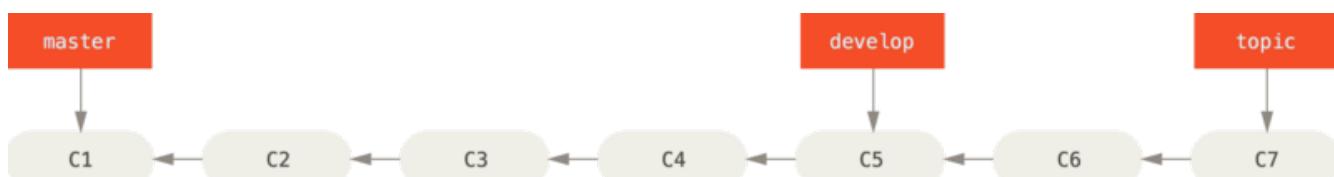


Figura 26. Una vista lineal del ramificado progresivo estable

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.

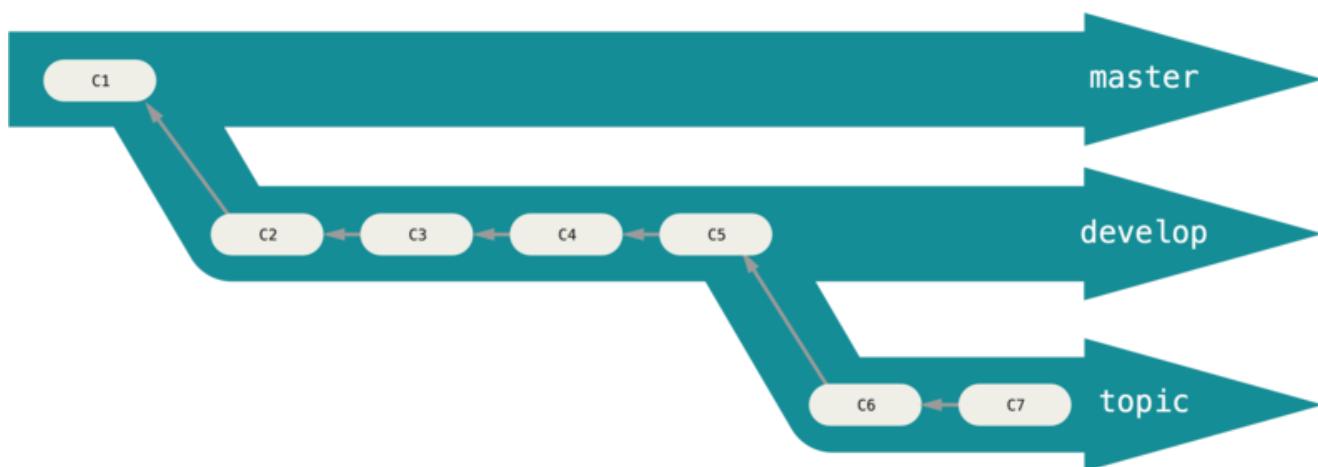


Figura 27. Una vista tipo "silo" del ramificado progresivo estable

Este sistema de trabajo se puede ampliar para diversos grados de estabilidad. Algunos proyectos muy grandes suelen tener una rama denominada `propuestas` o `pu` (del inglés “proposed updates”, propuesta de actualización), donde suele estar todo aquello que es

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. 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:

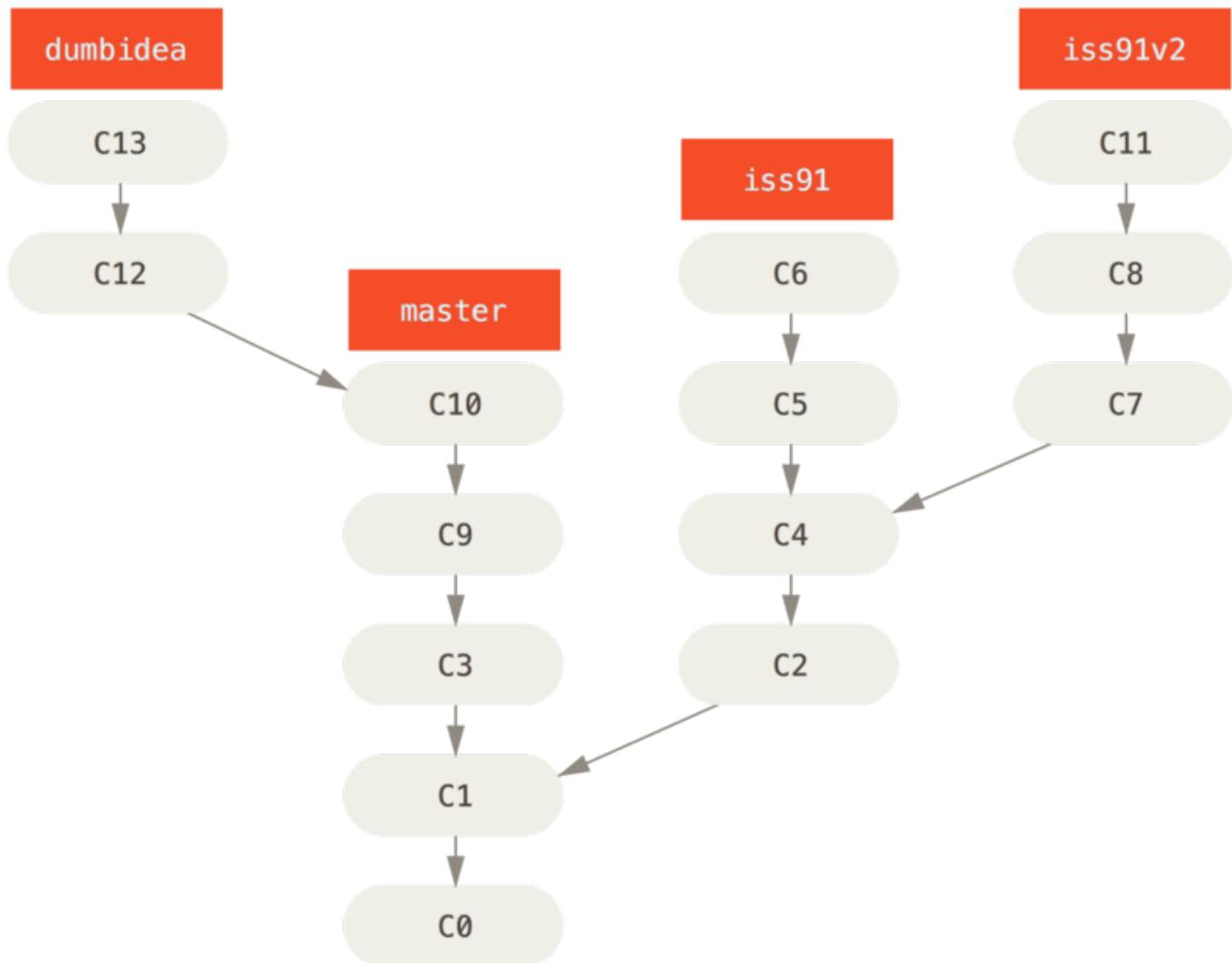


Figura 28. Múltiples ramas puntuales

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:

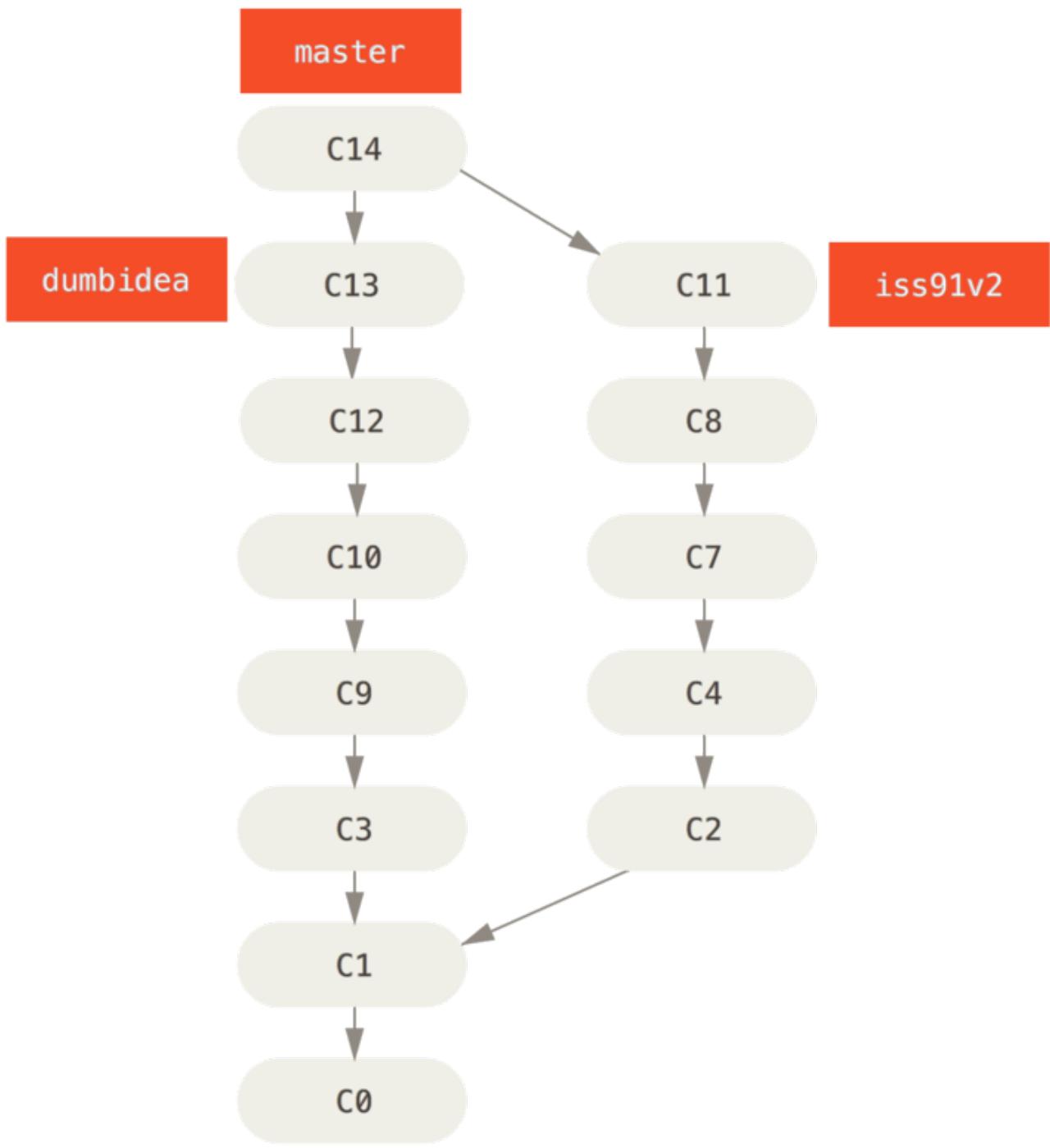


Figura 29. El historial tras fusionar `dumbidea` e `iss91v2`

Hablaremos un poco más sobre los distintos flujos de trabajo de tu proyecto Git en [Git en entornos distribuidos](#), 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 clon 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.

“origin” no es especial

NOTA

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.

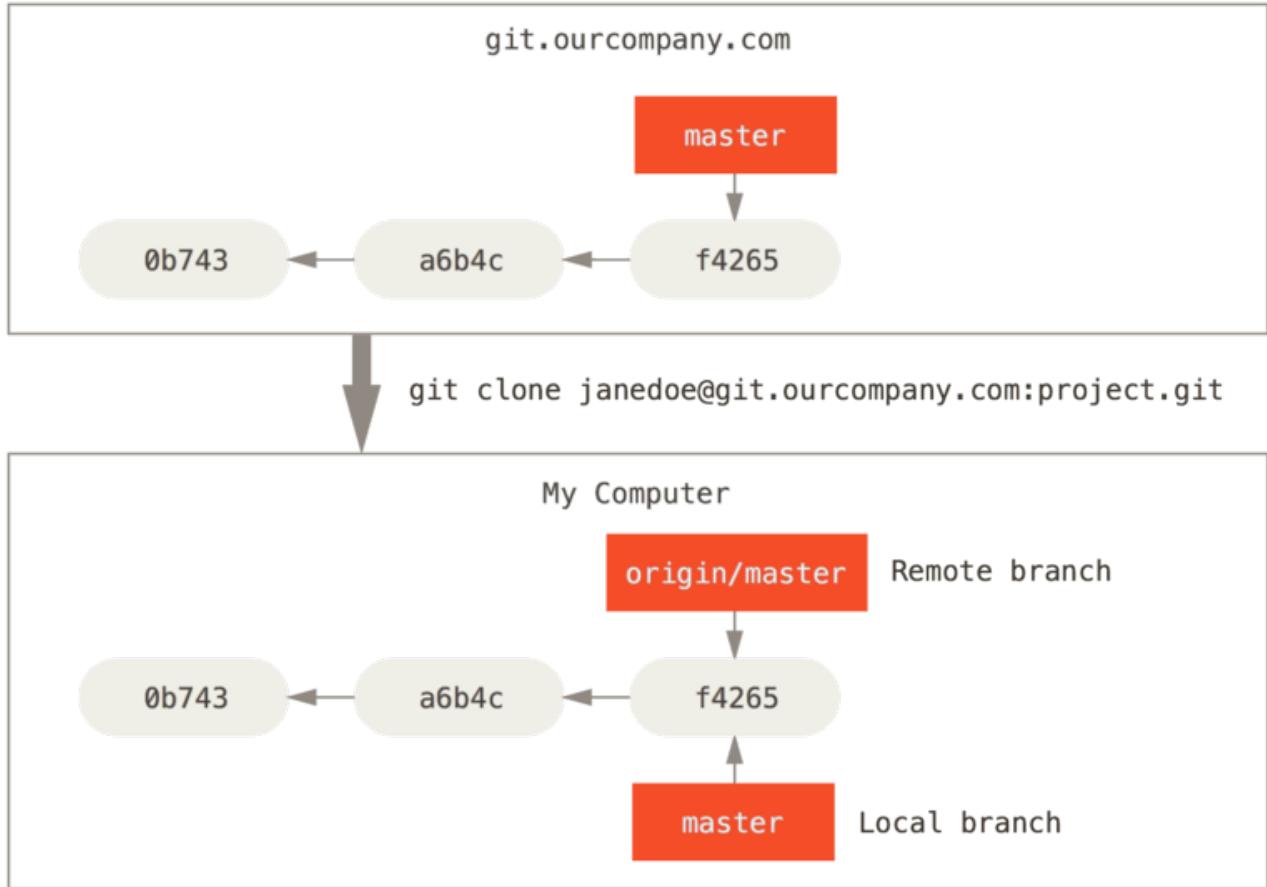


Figura 30. Servidor y repositorio local luego de ser clonado

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.

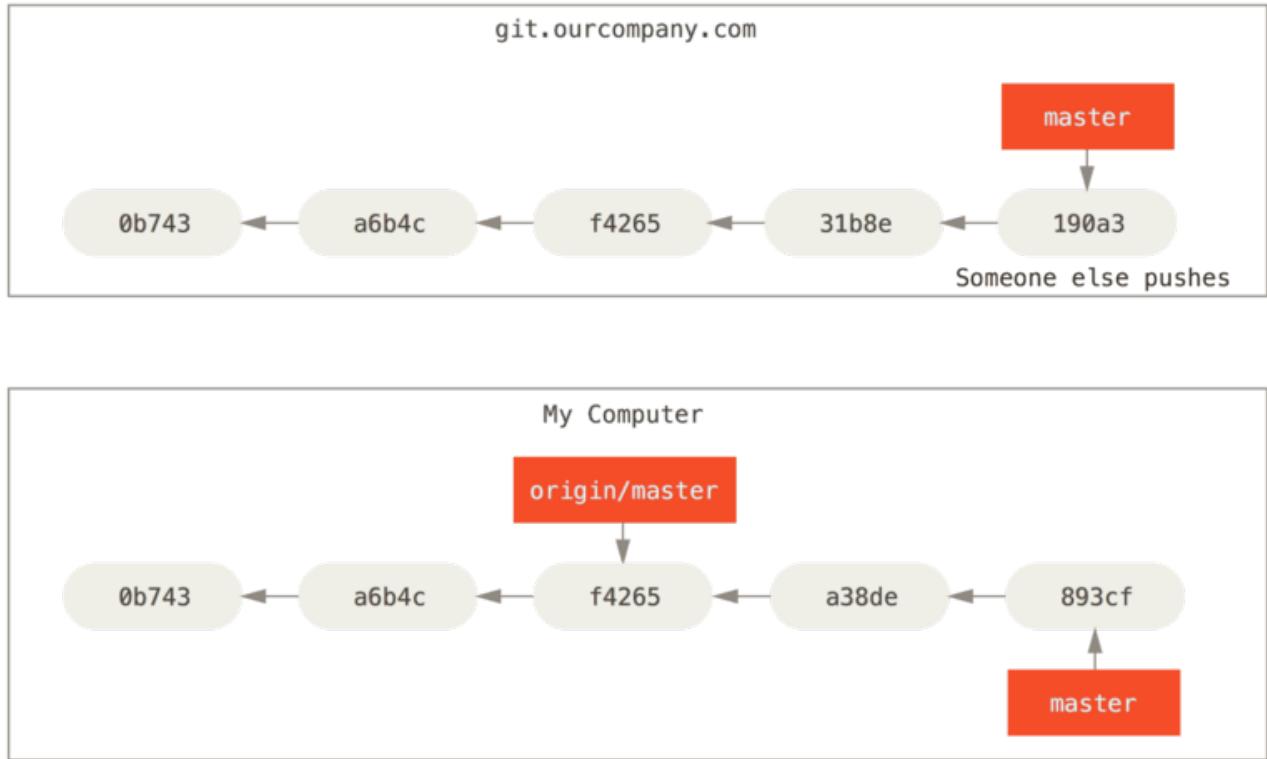


Figura 31. El trabajo remoto y el local pueden diverger

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.

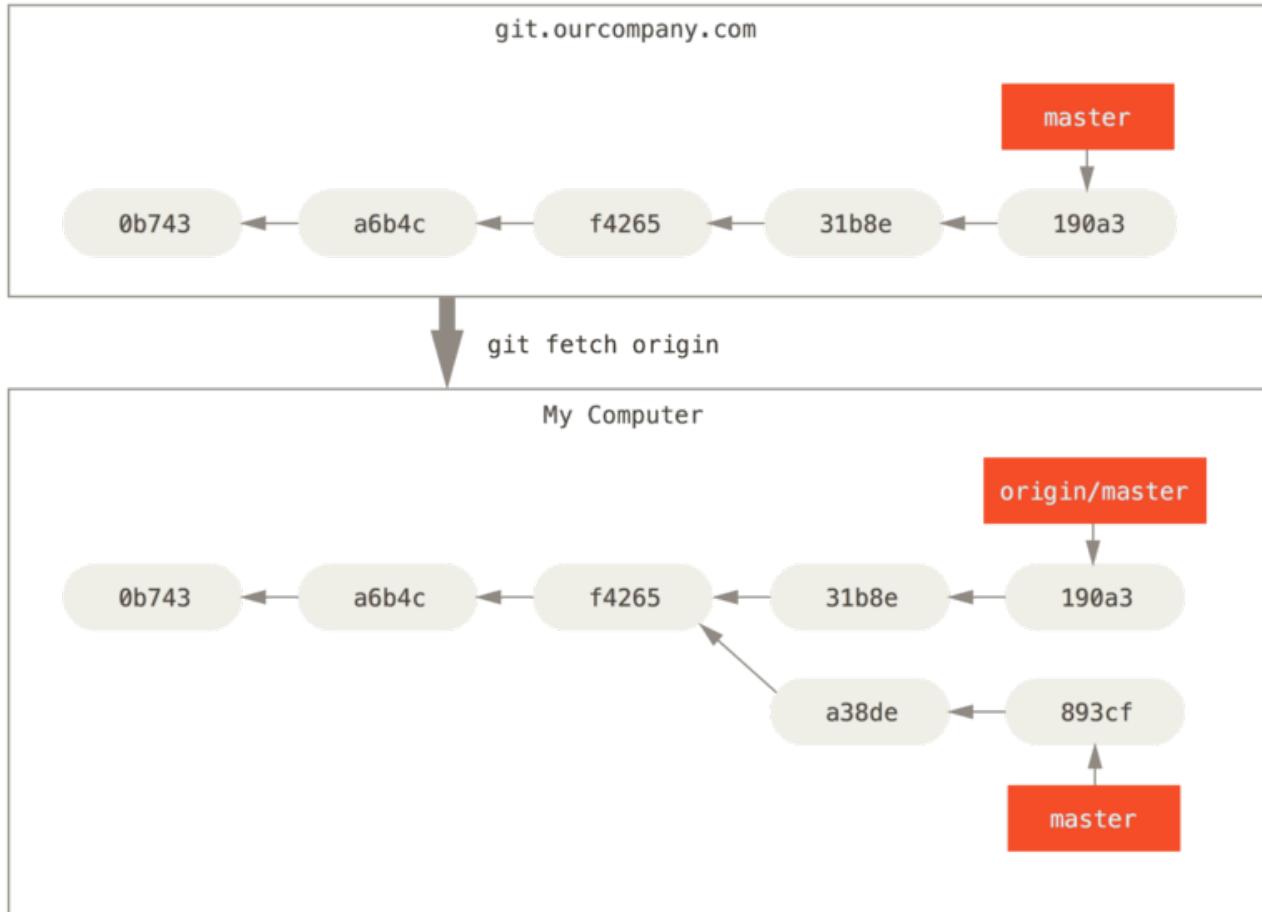


Figura 32. `git fetch` actualiza las referencias de tu remoto

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 [Fundamentos de Git](#). Puedes denominar `teamone` a este remoto al asignarle este nombre a la URL.

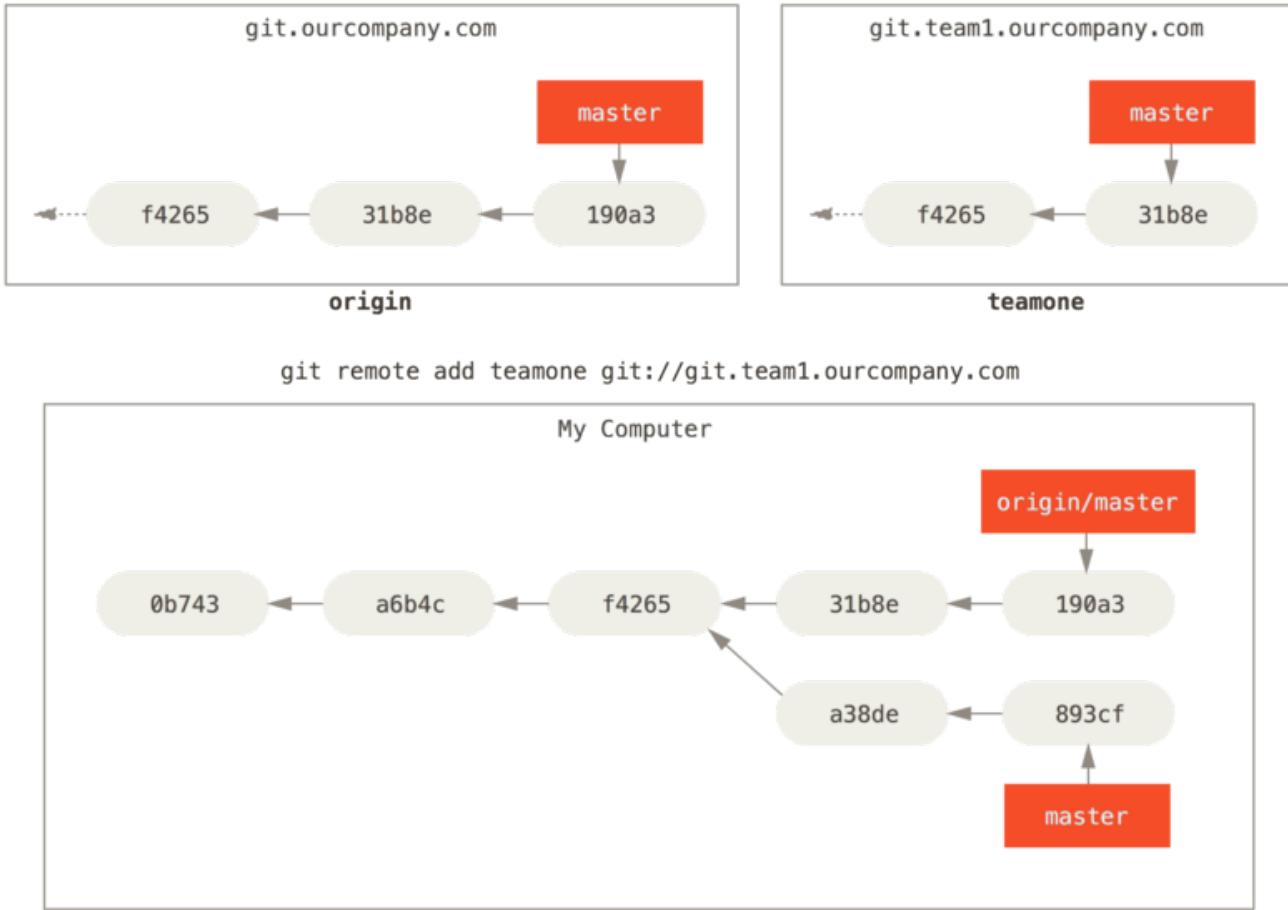


Figura 33. Añadiendo otro servidor como remoto

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`.

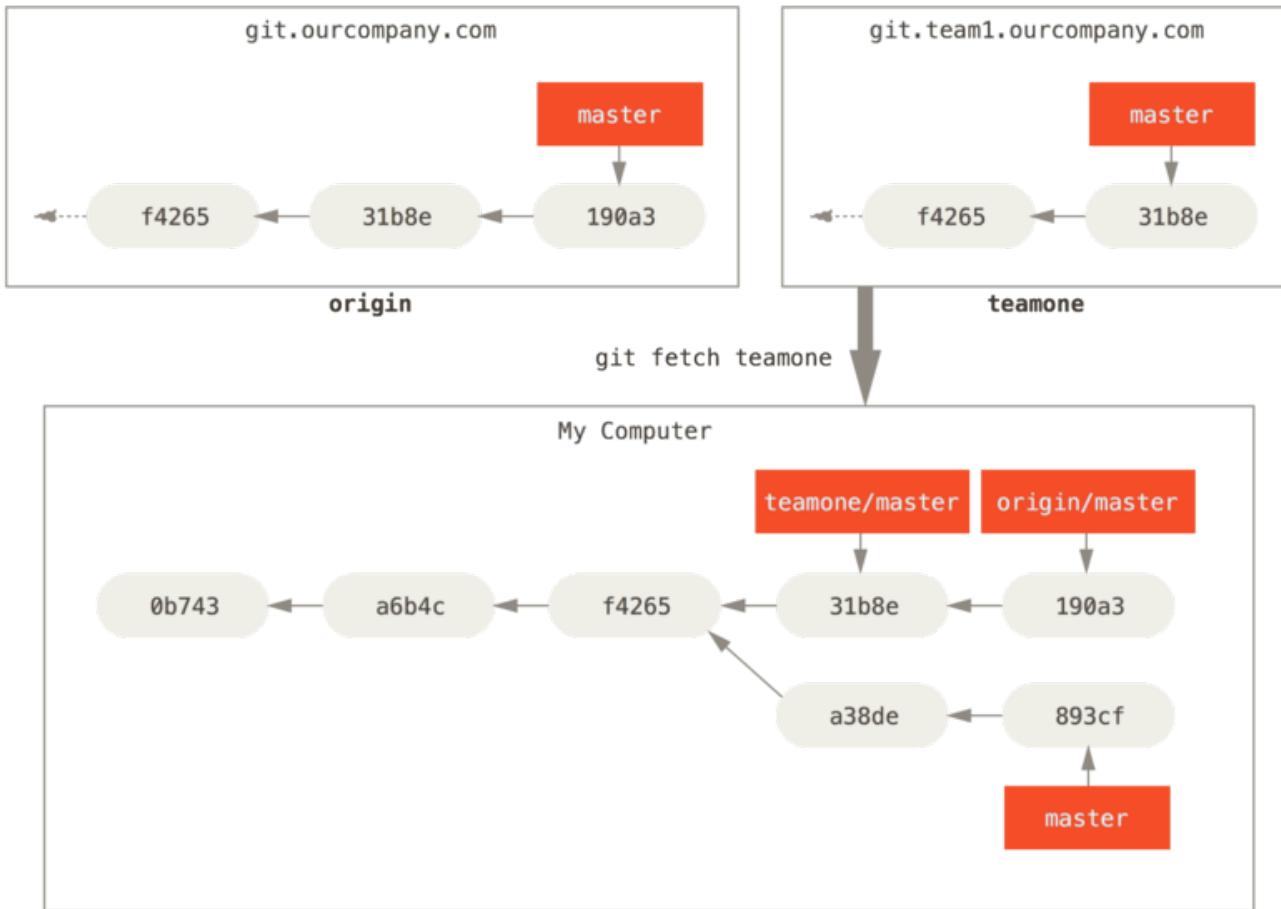


Figura 34. Seguimiento de la rama remota a través de `teamone/master`

Publicar

Cuando quieras 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 [Los entresijos internos de Git](#); 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 `awesombranch` 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.

NOTA

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 [Almacenamiento de credenciales](#).

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.
```

NOTA

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 quieras ver las ramas de seguimiento que tienes asignadas, 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 quieras tener los cambios por delante y por detrás actualizados, debes traértelos (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 que no tienes del servidor, 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.

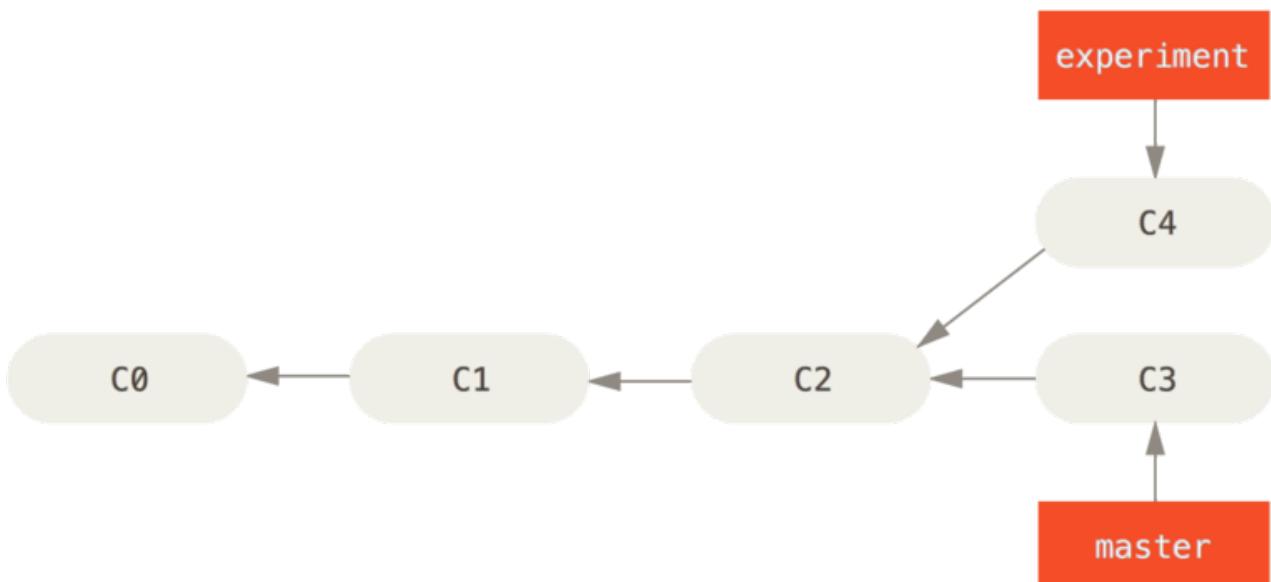


Figura 35. El registro de confirmaciones inicial

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).

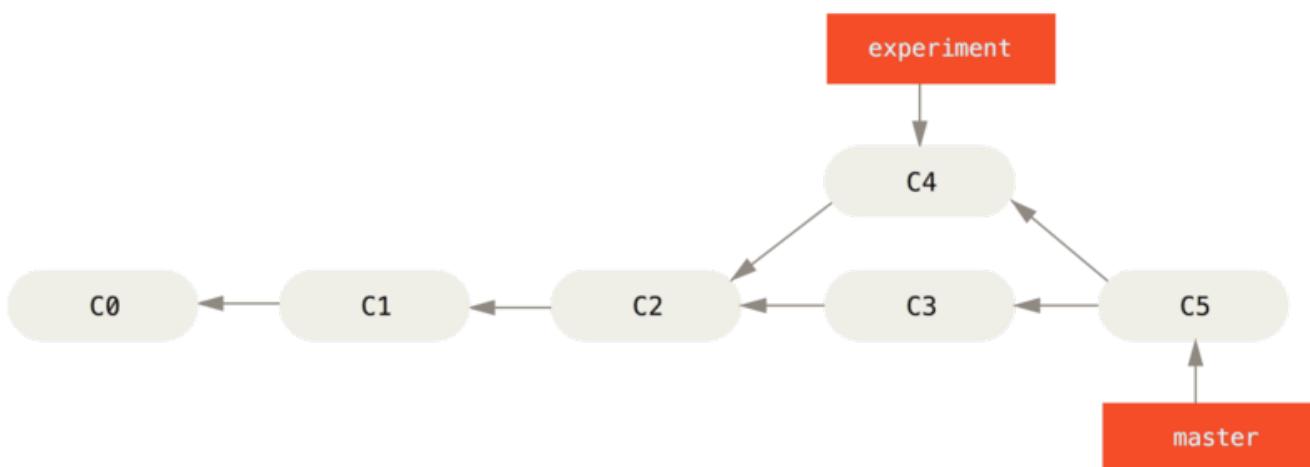


Figura 36. Fusionar una rama para integrar el registro de trabajos divergentes

Sin embargo, también hay otra forma de hacerlo: puedes capturar los cambios introducidos en C4 y reaplicarlos encima de C3. Esto es lo que en Git llamamos *reorganizar* (*rebasing*, en inglés). Con el comando `git rebase`, puedes capturar 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 quieras 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 que la rama de donde quieras reorganizar, y finalmente, vuelva a aplicar ordenadamente los cambios.

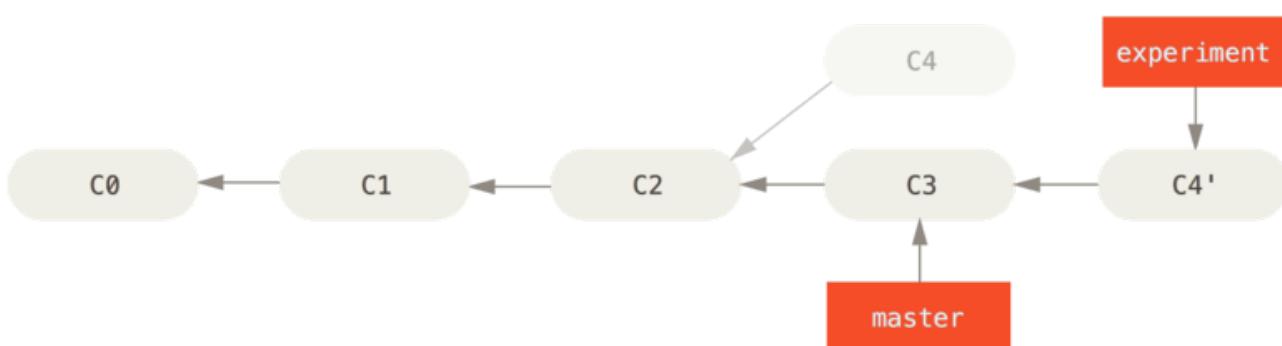


Figura 37. Reorganizando sobre C3 los cambios introducidos en C4

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
```

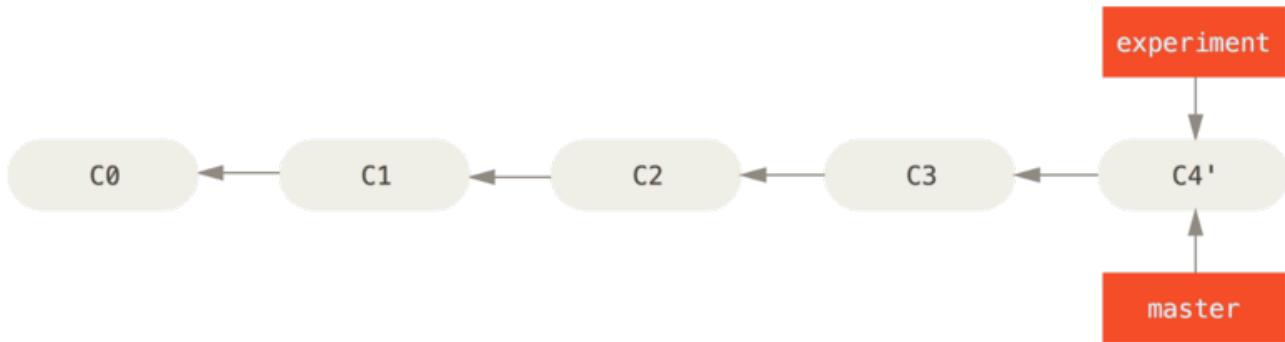


Figura 38. Avance rápido de la rama `master`

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 no 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 [Un historial con una rama puntual sobre otra rama puntual](#). 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.

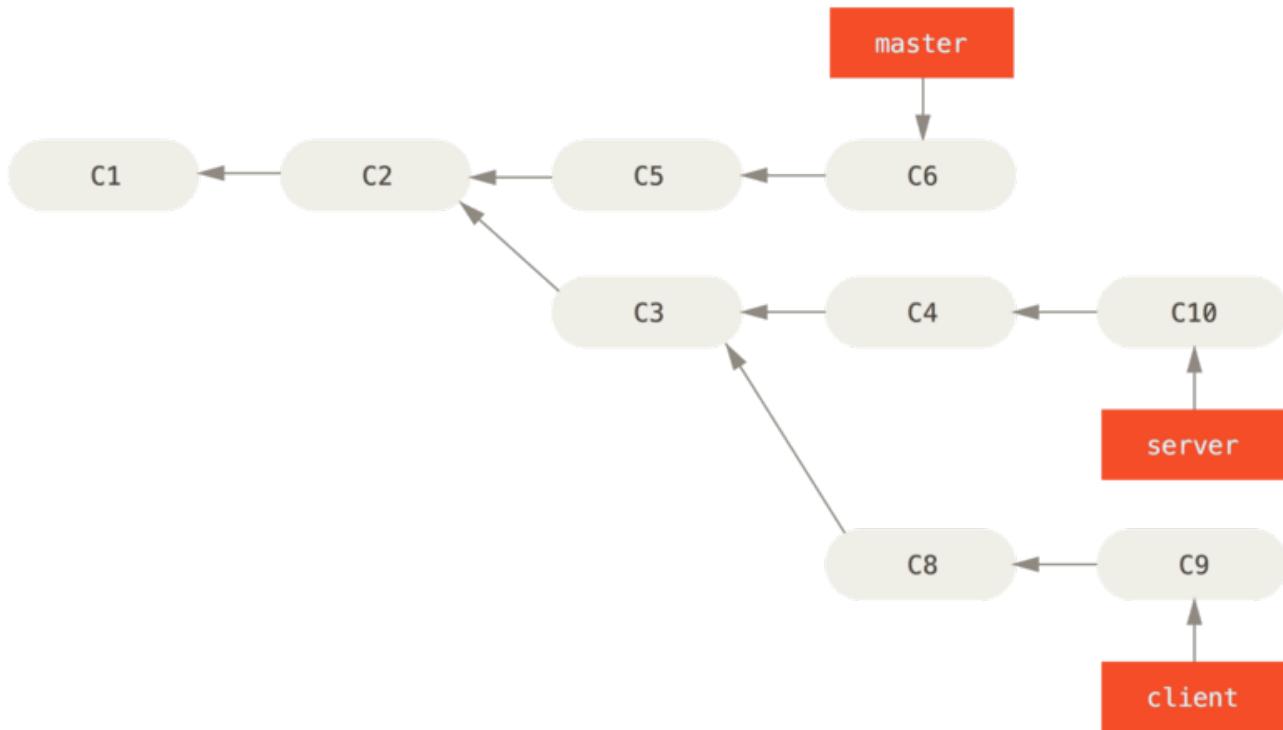


Figura 39. Un historial con una rama puntual sobre otra rama puntual

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.

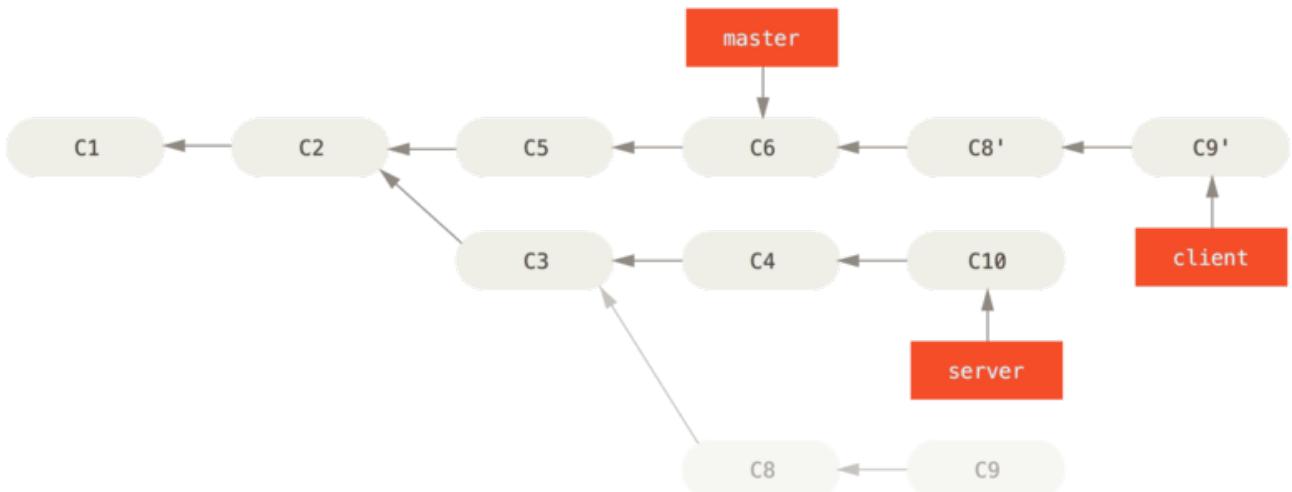


Figura 40. Reorganizando una rama puntual fuera de otra rama puntual

Y, tras esto, ya puedes avanzar la rama principal (ver [Avance rápido de tu rama master, para incluir los cambios de la rama client](#)):

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

Figura 41. 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 [Reorganizando la rama server sobre la rama master](#).

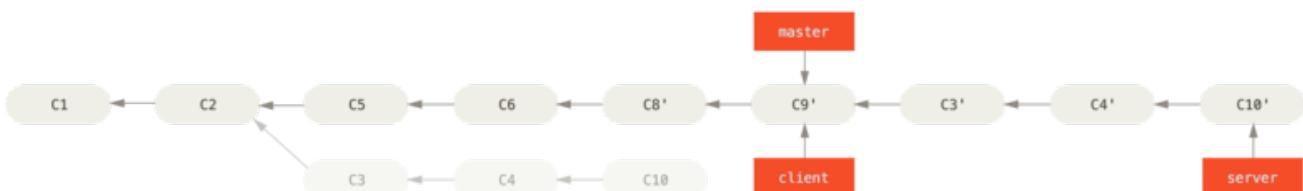


Figura 42. Reorganizando la rama server sobre la rama master

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 [Historial final de confirmaciones de cambio](#):

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



Figura 43. 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:

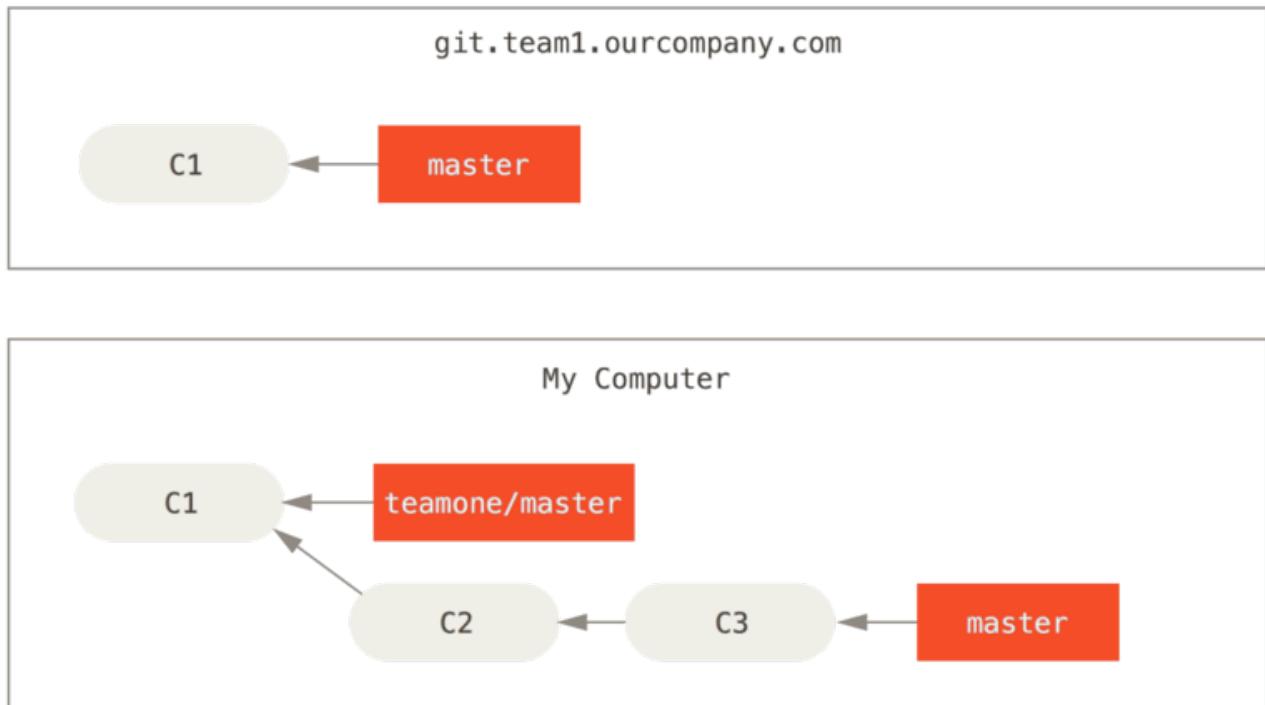


Figura 44. Clonar un repositorio y trabajar sobre él

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:

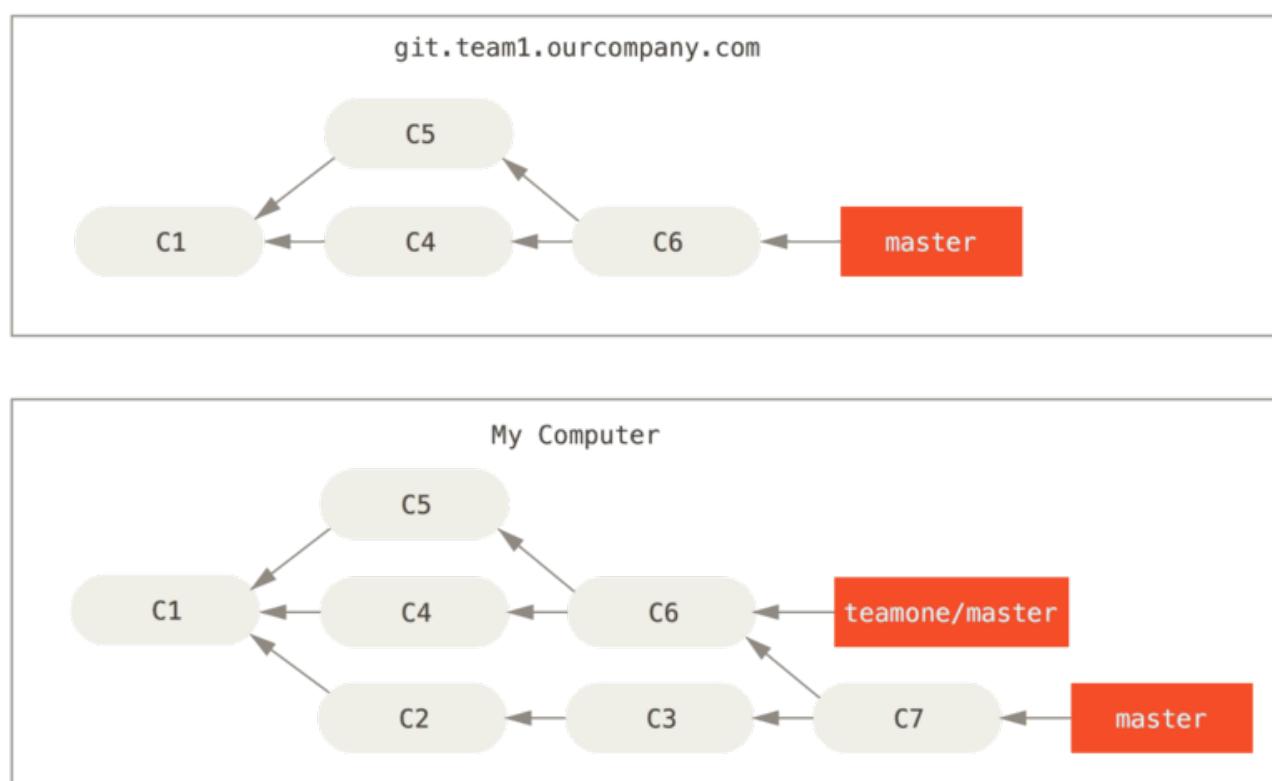


Figura 45. Traer (fetch) algunas confirmaciones de cambio (commits) y fusionarlas (merge) sobre tu trabajo

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.

Figura 46. Alguien envió (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:

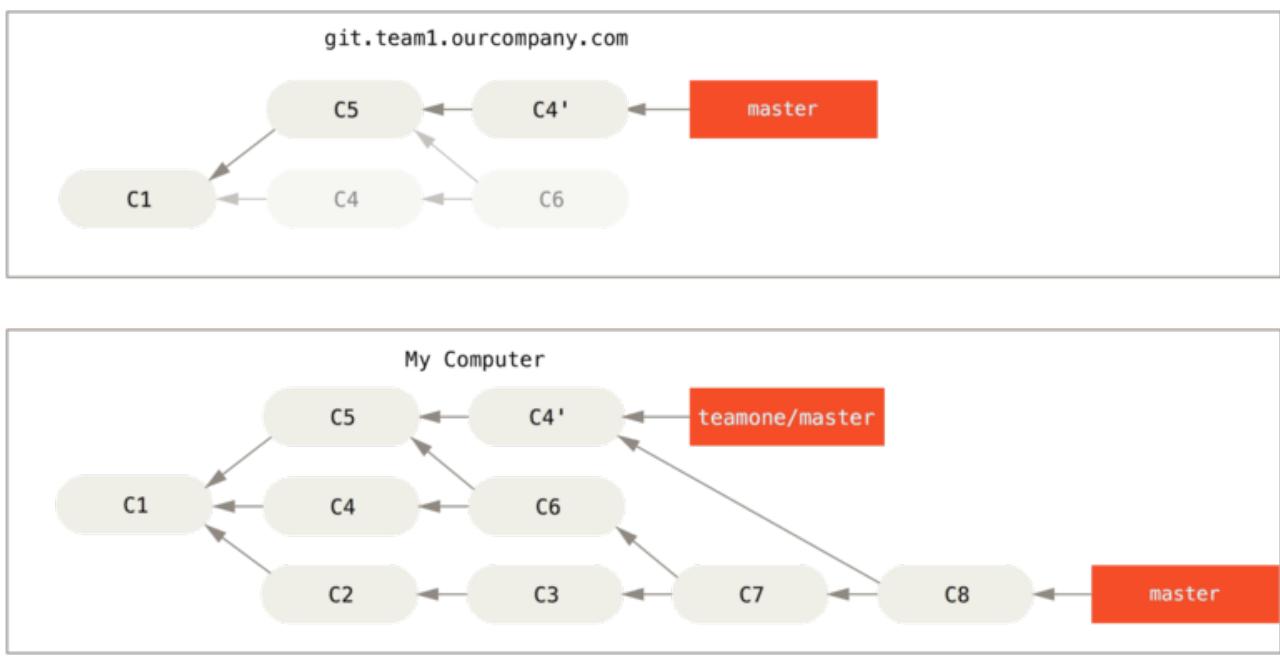


Figura 47. Vuelves a fusionar el mismo trabajo en una nueva fusión confirmada

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

Alguien envió (push) confirmaciones (commits) reorganizadas, abandonando las confirmaciones en las que tu habías basado tu trabajo 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 [Vuelves a fusionar el mismo trabajo en una nueva fusión confirmada](#), terminaremos con algo más parecido a [Reorganizar encima de un trabajo sobreescrito reorganizado..](#)

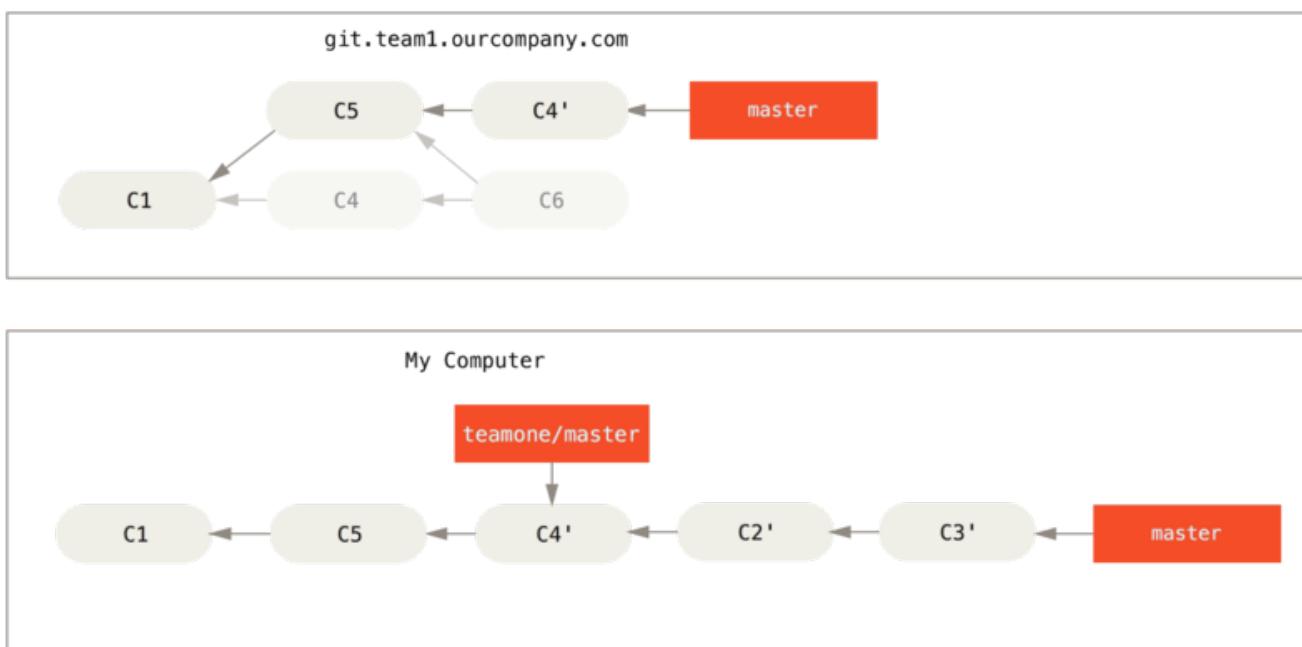


Figura 48. Reorganizar encima de un trabajo sobreescrito reorganizado.

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` yquieres 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 sólo 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 fué como ocurrió y el repositorio debería tener un registro de esto para la posteridad.

La otra forma de verlo puede ser 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 si es mejor 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 aún 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

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 distribuido.

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 computador. 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 [Los entresijos internos de Git](#) 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 tú 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 envíe (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 ejecuta sobre puertos estándar HTTP/S y puede 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\[\]](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 archivos

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 ese momento, cualquiera con acceso al servidor web donde se publique el repositorio podrá también clonarlo. Para permitir acceso de 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 la 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 archivos estáticos simples (véase [Los entresijos internos de Git](#) 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 sólo cuando se necesite, hace las cosas muy fáciles para el usuario final. Permitir 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.

Si 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 [Almacenamiento de credenciales](#) para ver cómo configurar el cacheo seguro de contraseñas HTTP en tu sistema.

El Protocolo 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 habilitarlo. Además, SSH es un protocolo de red autenticado 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 algún mecanismo de autenticación, cualquiera que encuentre la URL a tu proyecto en Internet, podrá enviar (push) contenidos a él. Ni qué decir tiene, que esto sólo 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

El principal problema 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 el 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.

NOTA

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 de 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 tú 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 tú 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 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 tendrá 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 están 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 ampliarlo con 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 ti. 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 tengan 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 engoroso. 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 es 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 autenticación a través de claves públicas SSH. Y, para ello, cada usuario del sistema ha de generarse una, si es que ya no la tiene. El proceso para hacerlo es similar en casi cualquier sistema operativo. Ante todo, asegúrate 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 ya tienes unas claves, simplemente situándote 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 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 enviándoselos por correo electrónico. La clave pública será una serie de números, letras y signos, algo así como esto:

```
$ cat ~/.ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAQEAk1OUpkDHrfHY17SbrmTIpNLTGK9Tjom/BWDSU
GPL+nafz1HDTYW7hdI4yZ5ew18JH4JW9jbhUFrviqzM7x1ELEVf4h91FX5QVkbPppSwg0cda3
Pbv7k0dJ/MTyB1WXFCR+HAo3FXRitBqxix1nKhXpHAZsMcilq8V6RjsNAQwdsdMFvSlVK/7XA
t3FaoJoAsncM1Q9x5+3V0Ww68/eIFmb1zuUFljQJKprrx88XypNDvjYNby6vw/Pb0rwert/En
mZ+AW40ZPnPnTPI89ZPmVMLuayrD2cE86Z/i18b+gw3r3+1nKatmIkjn2so1d01QraTlMqVSsbx
NrRFi9wrf+M7Q== schacon@mylaptop.local
```

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 estándar 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
Sdfd8AcCIicTDWbqLAcU4UpkaX8KyG1LwsNuuGztobF8m72ALC/nLF6JLtPofwFB1gc+myiv
07TCUSBdLQlgMVOfq1I2uPWQ0kOWQAHukE0mfjy2jctxSDBQ220ymjaNsHT4kgfZg2AYYgPq
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 archivos):

```
# 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 archivo `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 archivo con el camino (path)
completo
```

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: 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 computadores (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 “demonio” 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 cuales de tus repositorios ha de permitir acceso sin autenticar. Lo puedes hacer creando en cada repositorio un archivo llamado `git-daemon-export-ok`.

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

La presencia de este archivo dice a Git que este proyecto se puede servir sin problema sin necesidad de autentificación 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, se 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 archivo `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 archivo `.htaccess` que contenga las contraseñas cifradas de todos los usuarios válidos. Por ejemplo, para añadir el usuario “schacon” a este archivo:

```
$ 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 autenticación 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.

NOTA

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. At the top, there's a navigation bar with links for 'projects / .git / summary'. On the right side of the header, there are buttons for 'commit' (with a dropdown arrow), 'search' (with a magnifying glass icon), and a 're' checkbox. Below the header, there's a section titled 'description' containing the text: 'Unnamed repository; edit this file 'description' to name the repository.' It also shows 'owner' as Ben Straub and 'last change' as Wed, 11 Jun 2014 12:20:23 -0700 (21:20 +0200). A 'shortlog' section follows, listing commits from June 2014. Each commit entry includes the author, date, subject, and a link to the commit details. Below the shortlog is a 'tags' section, which lists various Git version tags with their corresponding commit dates and shortlog links.

Figura 49. The GitWeb web-based user interface.

Si quieras 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
```

Si quieres disponer permanentemente de un interfaz 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.

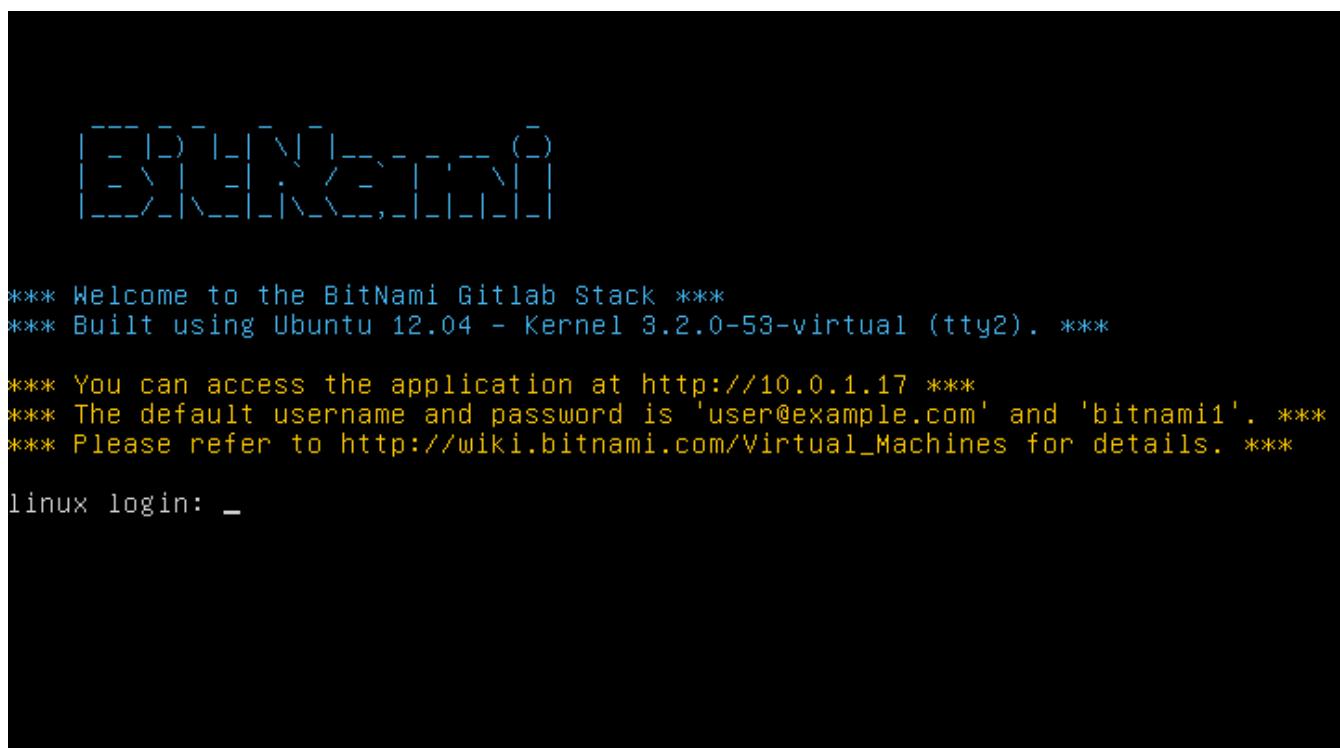


Figura 50. Página de login de la máquina virtual Bitnami.

Para las demás cosas, utiliza como guía los archivos `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@localhost`, 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.

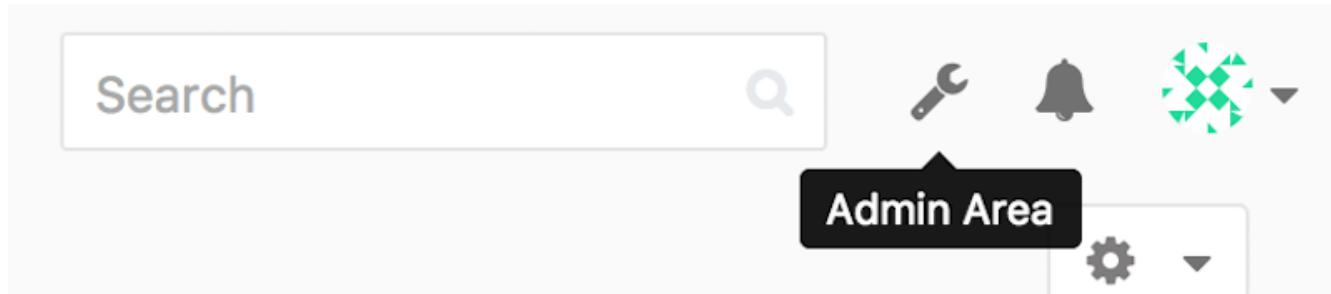


Figura 51. El ícono “Admin area” del menú de GitLab.

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`.

A screenshot of the GitLab User Administration interface. The top navigation bar includes "Admin Area", "Overview", "Monitoring", "Messages", "System Hooks", "Applications", "Abuse Reports", and a gear icon. Below this is a secondary navigation bar with "Overview", "Projects", "Users" (which is selected and highlighted in blue), "Groups", "Builds", and "Runners". A search bar at the top allows searching by name, email or username, with a "Name" dropdown and a "New User" button. The main content area displays a list of users under the heading "Active 26". Each user entry includes a profile picture, the user's name, their role (e.g., "Administrator", "Admin"), their email address, and a status message like "It's you!". To the right of each entry are "Edit" and "More" (dropdown) buttons. The user list includes entries for "Administrator Admin", "Betsy Rutherford II", "Brenden Hayes", "Cassandra Kilback", "Cathryn Leffler DVM", "Cecil Medhurst", "Dr. Joany Fisher", and "Jazmin Sipes".

Figura 52. 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 archivos. Todos los proyectos y datos de su espacio de nombres se perderán, así como cualquier grupo que le pertenezca. Esto es, por supuesto, la acción más permanente, 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>.

The screenshot shows the GitLab web interface for managing groups. At the top, there's a navigation bar with 'GitLab.org' on the left and a search bar on the right. Below the navigation is a header for the '@gitlab-org' group, featuring a logo of a stylized orange fox, the group name, a subtitle 'Open source software to collaborate on code', and buttons for 'Leave group' and 'Global'. The main content area displays a list of projects under the 'All Projects' tab. Each project entry includes a small icon, the project name, a brief description, and two circular icons on the right. The projects listed are:

- GitLab Development Kit: Get started with GitLab Rails development
- kubernetes-gitlab-demo: Idea to Production GitLab Demo running on Kubernetes
- omnibus-gitlab: This project creates full-stack platform-specific downloadable packages for GitLab.
- GitLab Enterprise Edition: GitLab Enterprise Edition
- gitlab-shell: SSH access and repository management app for GitLab
- gitlab-ci-multi-runner: GitLab Runner

Figura 53. Pantalla de administración de grupos en GitLab.

Cada grupo se asocia con un conjunto de usuarios, donde 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 los 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 no 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 petición HTTP POST con determinados datos JSON cuando ocurran ciertos 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 archivos 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 de colaboración, más desacoplada, 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 a su vez la cooperación de usuarios a 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 archivo 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 sin 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 [GitHub](#), 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 ti y para tu organización.

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 ti como para el responsable del proyecto, y también como mantener adecuadamente un proyecto con múltiples desarrolladores.

Flujos de trabajo distribuidos

A diferencia de los Sistemas Centralizados de Control de Versiones (CVCSSs, Centralized Version Control Systems), la naturaleza distribuida 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 código y todos sincronizan su trabajo con él. Unos cuantos desarrolladores son nodos de trabajo - consumidores de dicho repositorio - y sincronizan con ese punto.

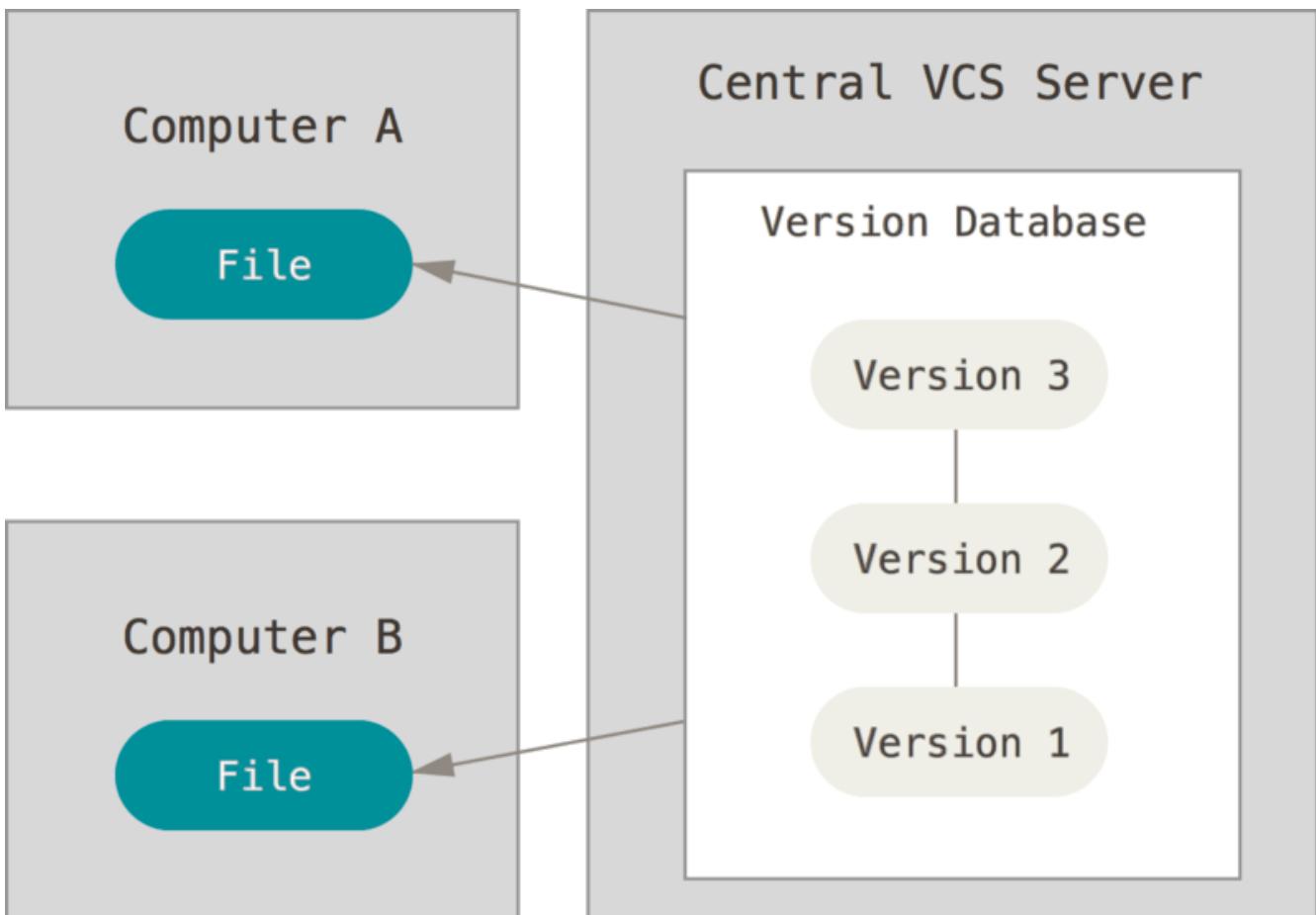


Figura 54. Centralized workflow.

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 sobrescribir los cambios del primero. Este concepto es válido tanto en Git como en Subversion.

Si ya está cómodo con un flujo de trabajo centralizado en su empresa o en su equipo, puede seguir utilizando fácilmente ese flujo de trabajo con Git. Simplemente configure un único repositorio, y dé a cada uno en su equipo acceso de empuje; Git no permitirá que los usuarios se sobrescriban entre sí. Digamos que John y Jessica empiezan a trabajar al mismo tiempo. John termina su cambio y lo empuja al servidor. Entonces Jessica intenta empujar sus cambios, pero el servidor los rechaza. Le dice que está tratando de empujar cambios no rápidos y que no podrá hacerlo hasta que busque y se fuse. Este flujo de trabajo es atractivo para mucha gente porque es un paradigma con el que muchos están familiarizados y cómodos.

Esto tampoco se limita a los equipos pequeños. Con el modelo de ramificación de Git, es posible que cientos de desarrolladores trabajen con éxito en un único proyecto a través de docenas de ramas simultáneamente.

Flujo de Trabajo Administrador-Integración

Debido a que Git permite tener múltiples repositorios remotos, es posible tener un flujo de trabajo donde cada desarrollador tenga acceso de escritura a su propio repositorio público y acceso de lectura a todos los demás. Este escenario a menudo incluye un repositorio canónico que representa el proyecto "oficial". Para contribuir a ese proyecto, creas tu propio clon público del proyecto y haces pull con tus cambios. Luego, puedes enviar una solicitud al administrador del proyecto principal para que agregue los cambios. Entonces, el administrador agrega el repositorio como remoto, prueba los cambios localmente, los combina en su rama y los envía al repositorio. El proceso funciona de la siguiente manera. (ver [Flujo de Trabajo Administrador-Integración](#).):

1. El administrador del proyecto hace un push al repositorio público.
2. El contribuidor clona ese repositorio y realiza los cambios.
3. El contribuidor realiza un push con su copia pública del proyecto.
4. El contribuidor envía un correo electrónico al administrador pidiendo que haga pull de los cambios.
5. El administrador agrega el repositorio del contribuidor como remoto y fusiona ambos localmente.
6. El administrador realiza un push con la fusión del código al repositorio principal.

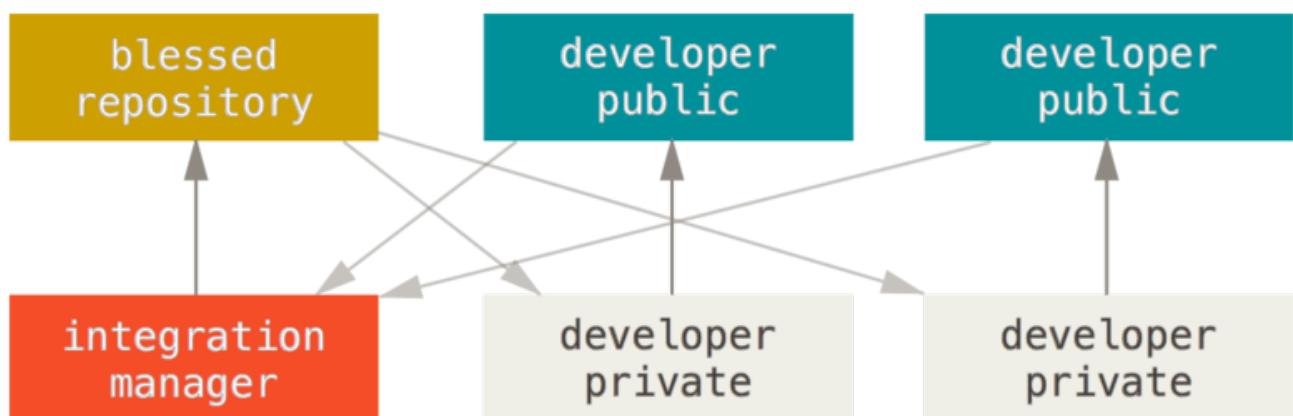


Figura 55. Flujo de Trabajo Administrador-Integración.

Este es un flujo de trabajo muy común con herramientas basadas en hubs como GitHub o GitLab, donde es fácil hacer un fork de un proyecto e introducir los cambios en este fork para que todos puedan verlos. Una de las principales ventajas de este enfoque es que el contribuidor puede continuar realizando cambios y el administrador principal del repositorio puede incorporar los cambios en cualquier momento. Los contribuidores no tienen que esperar a que el proyecto incorpore sus cambios; cada parte puede trabajar a su propio ritmo.

Flujo de Trabajo Dictador-Tenientes

Esta es una variante de un flujo de trabajo de múltiples repositorios. Generalmente es utilizado por grandes proyectos con cientos de colaboradores; Un ejemplo famoso es el kernel de Linux. Varios administradores de integración están a cargo de ciertas partes

del repositorio. Se les llaman “tenientes”. Todos los tenientes tienen un gerente de integración conocido como el “dictador benévolo”. El repositorio del dictador benevolente sirve como el repositorio de referencia del cual todos los colaboradores necesitan realizar pull. El proceso funciona así. (ver [Flujo de Trabajo Dictador Benevolente](#).):

1. Los desarrolladores trabajan en su propia rama específica y fusionan su código en la rama `master`, la cual, es una copia de la rama del dictador.
2. Los tenientes fusionan el código de las ramas `master` de los desarrolladores en sus ramas `master` de tenientes.
3. El dictador fusiona la rama `master` de los tenientes a su rama `master` de dictador.
4. El dictador hace push del contenido de su rama `master` al repositorio para que otros fusionen los cambios a sus ramas.

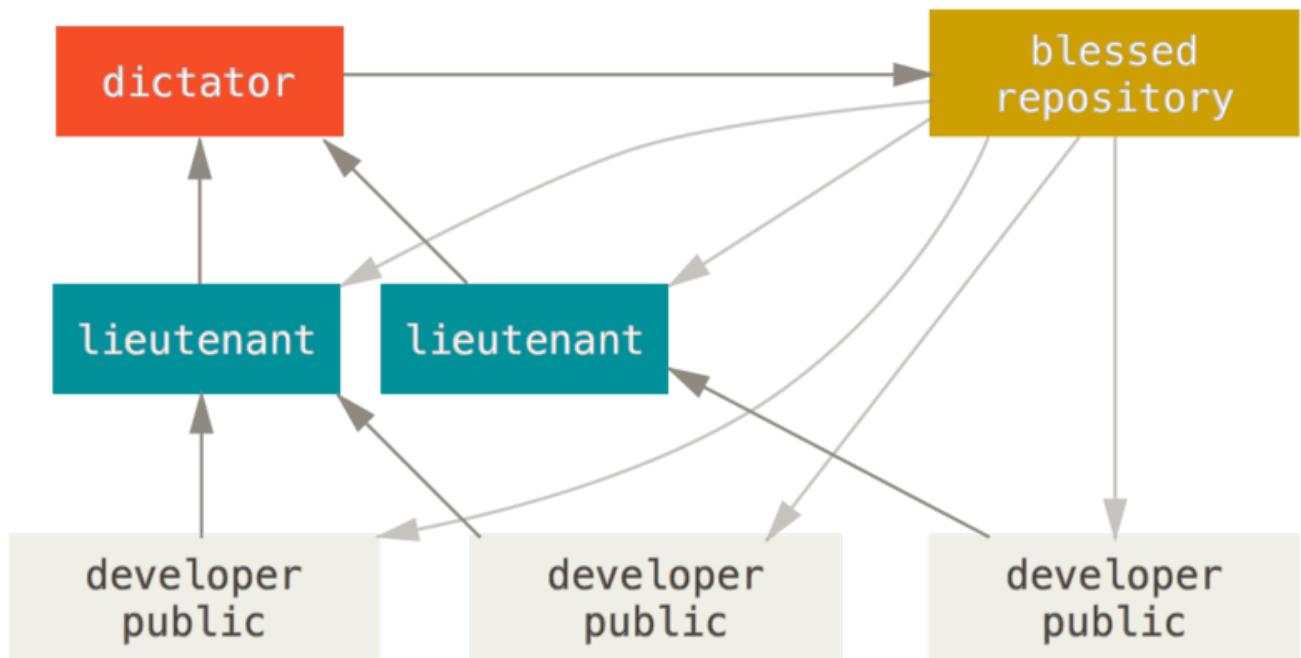


Figura 56. Flujo de Trabajo Dictador Benevolente.

Este tipo de flujo de trabajo no es común, pero puede ser útil en proyectos muy grandes o en entornos altamente jerárquicos. Permite al líder del proyecto (el dictador) delegar gran parte del trabajo y recopilar grandes subconjuntos de código en múltiples puntos antes de integrarlos.

Resumen de Flujos de Trabajo

Estos son algunos de los flujos de trabajo de uso común que son posibles con un sistema distribuido como Git, pero se puede observar que hay muchas posibles variaciones que buscan adaptarse a tu flujo de trabajo particular. Ahora puedes (con suerte) determinar qué combinación de flujo de trabajo puede funcionar mejor para ti, cubriremos algunos ejemplos más específicos sobre cómo cumplir los roles principales que conforman los diferentes flujos. En la siguiente sección, aprenderás sobre algunos patrones comunes para contribuir a un proyecto.

Contribuyendo a un Proyecto

La principal dificultad con la descripción de cómo contribuir a un proyecto es que hay una gran cantidad de variaciones sobre cómo se hace. Debido a que Git es muy flexible, las personas pueden trabajar juntas de muchas maneras, y es problemático describir cómo deberían contribuir: cada proyecto es un poco diferente. Algunas de las variables involucradas son conteo de contribuyentes activos, flujo de trabajo elegido, acceso de confirmación y posiblemente el método de contribución externa.

La primera variable es el conteo de contribuyentes activos: ¿cuántos usuarios están contribuyendo activamente al código de un proyecto y con qué frecuencia? En muchos casos, tendrá dos o tres desarrolladores con algunos commits por día o posiblemente menos para proyectos un tanto inactivos. Para empresas o proyectos más grandes, la cantidad de desarrolladores podría ser de miles, con cientos o miles de compromisos cada día. Esto es importante porque con más y más desarrolladores, se encontrará con más problemas para asegurarse de que su código se aplique de forma limpia o se pueda fusionar fácilmente. Los cambios que envíe pueden quedar obsoletos o severamente interrumpidos por el trabajo que se fusionó mientras estaba trabajando o mientras los cambios estaban esperando ser aprobados o aplicados. ¿Cómo puede mantener su código constantemente actualizado y sus confirmaciones válidas?

La siguiente variable es el flujo de trabajo en uso para el proyecto. ¿Está centralizado, con cada desarrollador teniendo el mismo acceso de escritura a la línea de código principal? ¿El proyecto tiene un mantenedor o un gerente de integración que verifica todos los parches? ¿Están todos los parches revisados por pares y aprobados? ¿Está usted involucrado en ese proceso? ¿Hay un sistema de tenientes en su lugar, y tiene que enviar su trabajo primero?

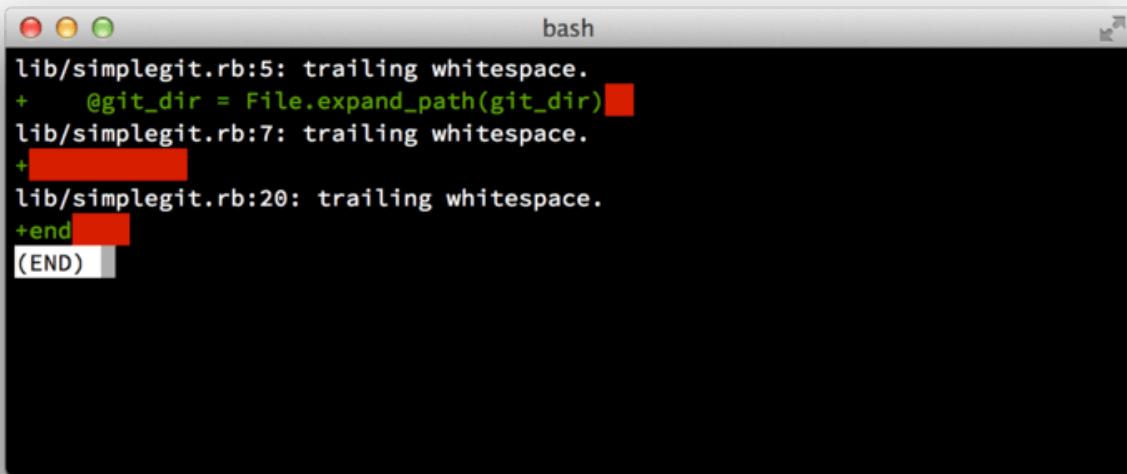
El siguiente problema es su acceso de confirmación. El flujo de trabajo requerido para contribuir a un proyecto es muy diferente si usted tiene acceso de escritura al proyecto que si no lo tiene. Si no tiene acceso de escritura, ¿cómo prefiere el proyecto aceptar el trabajo contribuido? ¿Incluso tiene una política? ¿Cuánto trabajo estás contribuyendo a la vez? ¿Con qué frecuencia contribuye?

Todas estas preguntas pueden afectar la forma en que se contribuye de manera efectiva a un proyecto y los flujos de trabajo preferidos o disponibles para usted. Cubriremos aspectos de cada uno de éstos en una serie de casos de uso, pasando de simples a más complejos; debería poder construir los flujos de trabajo específicos que necesita en la práctica a partir de estos ejemplos.

Pautas de confirmación

Antes de comenzar a buscar casos de uso específicos, aquí hay una nota rápida sobre los mensajes de confirmación. Tener una buena guía para crear compromisos y apegarse a ella hace que trabajar con Git y colaborar con otros sea mucho más fácil. El proyecto Git proporciona un documento que presenta una serie de buenos consejos para crear compromisos a partir de los cuales enviar parches: puede leerlos en el código fuente de Git en el archivo [Documentation / SubmittingPatches](#).

En primer lugar, no desea enviar ningún error de espacios en blanco. Git proporciona una manera fácil de verificar esto: antes de comprometerse, ejecute `git diff --check`, que identifica posibles errores de espacio en blanco y los enumera por usted.



```
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)
```

Figura 57. Output of `git diff --check`.

Si ejecuta ese comando antes de confirmar, puede ver si está a punto de cometer errores de espacio en blanco que pueden molestar a otros desarrolladores.

A continuación, intente hacer de cada commit un conjunto de cambios lógicamente separado. Si puede, trate de hacer que sus cambios sean digeribles: no codifique durante un fin de semana entero en cinco asuntos diferentes para luego enviarlos todos como un compromiso masivo el lunes. Incluso si no confirma durante el fin de semana, utilice el área de etapas el lunes para dividir su trabajo en al menos una confirmación por cuestión, con un mensaje útil por confirmación. Si algunos de los cambios modifican el mismo archivo, intente utilizar `git add --patch` para representar parcialmente los archivos (se detalla en << _ interactive_staging >>). La instantánea del proyecto en la punta de la rama es idéntica, ya sea que realice una confirmación o cinco, siempre que todos los cambios se agreguen en algún momento, así que trate de facilitar las cosas a sus compañeros desarrolladores cuando tengan que revisar sus cambios. Este enfoque también hace que sea más fácil retirar o revertir uno de los conjuntos de cambios si lo necesita más adelante. << _ rewriting_history >> describe una serie de trucos útiles de Git para reescribir el historial y organizar de forma interactiva los archivos: use estas herramientas para crear un historial limpio y comprensible antes de enviar el trabajo a otra persona.

Lo último a tener en cuenta es el mensaje de compromiso. Tener el hábito de crear mensajes de compromiso de calidad hace que usar y colaborar con Git sea mucho más fácil. Como regla general, sus mensajes deben comenzar con una sola línea que no supere los 50 caracteres y que describa el conjunto de cambios de forma concisa, seguido de una línea en blanco, seguida de una explicación más detallada. El proyecto Git requiere que la explicación más detallada incluya su motivación para el cambio

y contraste su implementación con el comportamiento anterior: esta es una buena guía a seguir. También es una buena idea usar el tiempo presente imperativo en estos mensajes. En otras palabras, use comandos. En lugar de `` agregué pruebas para ' o `` Añadir pruebas para ', use `` Agregar pruebas para ". Aquí hay una plantilla escrita originalmente por Tim Pope:

Resumen de cambios cortos (50 caracteres o menos)

Texto explicativo más detallado, si es necesario. Ajustarlo a aproximadamente 72 caracteres más o menos. En algunos contextos, la primera línea se trata como el tema de un correo electrónico y el resto de el texto como el cuerpo. La línea en blanco que separa el resumen del cuerpo es crítica (a menos que omita el cuerpo enteramente); herramientas como ``rebase'' pueden confundirse si ejecuta los dos juntos.

Otros párrafos vienen después de las líneas en blanco.

- Los puntos de viñetas también están bien
- Típicamente se usa un guión o asterisco para la viñeta, precedido por un solo espacio, con líneas en blanco entre viñetas, pero las convenciones varían aquí

Si todos sus mensajes de confirmación se ven así, las cosas serán mucho más fáciles para usted y para los desarrolladores con los que trabaja. El proyecto Git tiene mensajes de confirmación bien formateados. Intente ejecutar `git log --no-merges` allí para ver cómo se ve un historial de commit de proyecto muy bien formateado.

En los siguientes ejemplos y en la mayor parte de este libro, en aras de la brevedad, no verá mensajes con un formato agradable como éste; en cambio, usamos la opción `-m` para `git commit`. Haz lo que decimos, no como lo hacemos.

Pequeño equipo privado

La configuración más simple que es probable encuentre, es un proyecto privado con uno o dos desarrolladores más. `` Privado '', en este contexto, significa fuente cerrada, no accesible para el mundo exterior. Usted y los demás desarrolladores son los únicos que tienen acceso de inserción al repositorio.

En este entorno, puede seguir un flujo de trabajo similar a lo que podría hacer al usar Subversion u otro sistema centralizado. Aún obtiene las ventajas de cosas como el compromiso fuera de línea y una bifurcación y fusión mucho más simples, pero el flujo de trabajo puede ser muy similar; la principal diferencia es que las fusiones ocurren en el lado del cliente en lugar del servidor en el momento de la confirmación. Veamos cómo se vería cuando dos desarrolladores comienzan a trabajar juntos con un repositorio compartido. El primer desarrollador, John, clona el repositorio, hace un cambio y se compromete localmente. (Los mensajes de protocolo se han reemplazado con

... en estos ejemplos para acortarlos un poco).

```
# 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(-)
```

El segundo desarrollador, Jessica, hace lo mismo: clona el repositorio y comete un cambio:

```
# 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(-)
```

Ahora, Jessica lleva su trabajo al servidor:

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

John intenta impulsar su cambio, también:

```
# 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 no puede presionar porque Jessica ha presionado mientras tanto. Esto es especialmente importante de entender si está acostumbrado a Subversion, porque notará que los dos desarrolladores no editaron el mismo archivo. Aunque Subversion realiza automáticamente una fusión de este tipo en el servidor si se editan diferentes archivos, en Git debe fusionar los commit localmente. John tiene que buscar los cambios de

Jessica y fusionarlos antes de que se le permita presionar:

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

En este punto, el repositorio local de John se ve así:

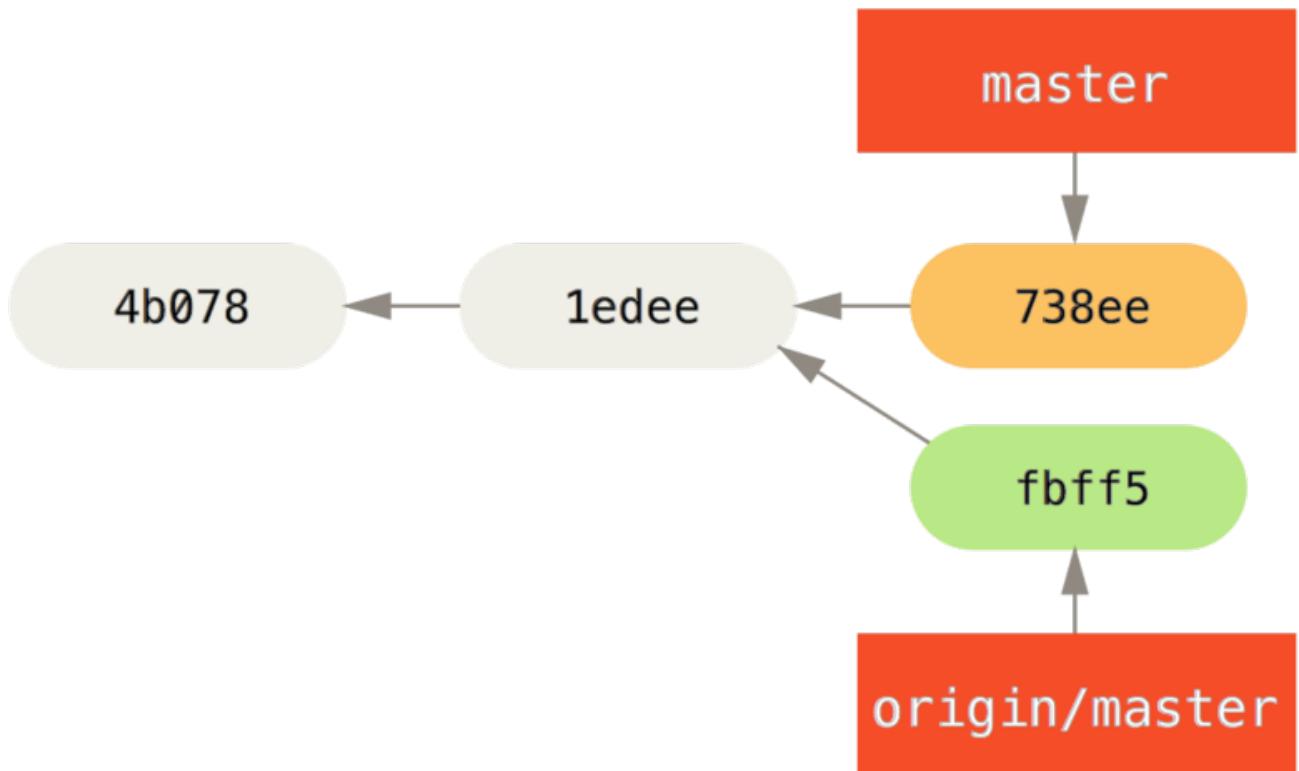


Figura 58. John's divergent history.

John tiene una referencia a los cambios que Jessica elevó, pero tiene que fusionarlos en su propio trabajo antes de que se le permita presionar:

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

La fusión funciona sin problemas: el historial de compromisos de John ahora se ve así:

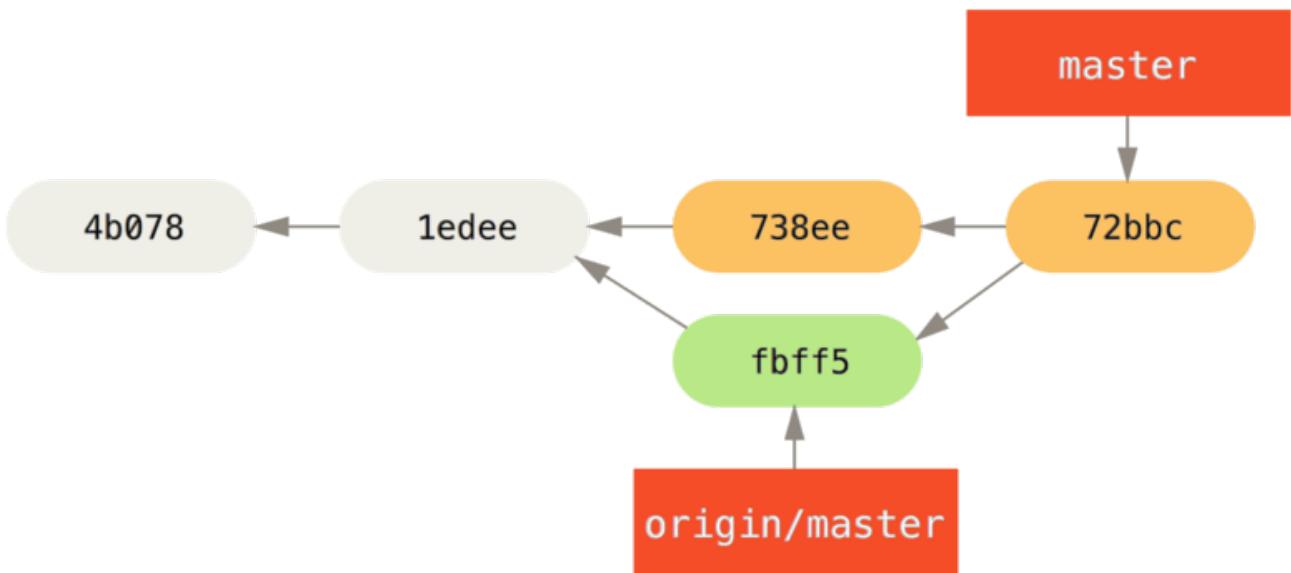


Figura 59. John's repository after merging `origin/master`.

Ahora, John puede probar su código para asegurarse de que todavía funciona correctamente, y luego puede enviar su nuevo trabajo combinado al servidor:

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

Finalmente, el historial de compromisos de John se ve así:

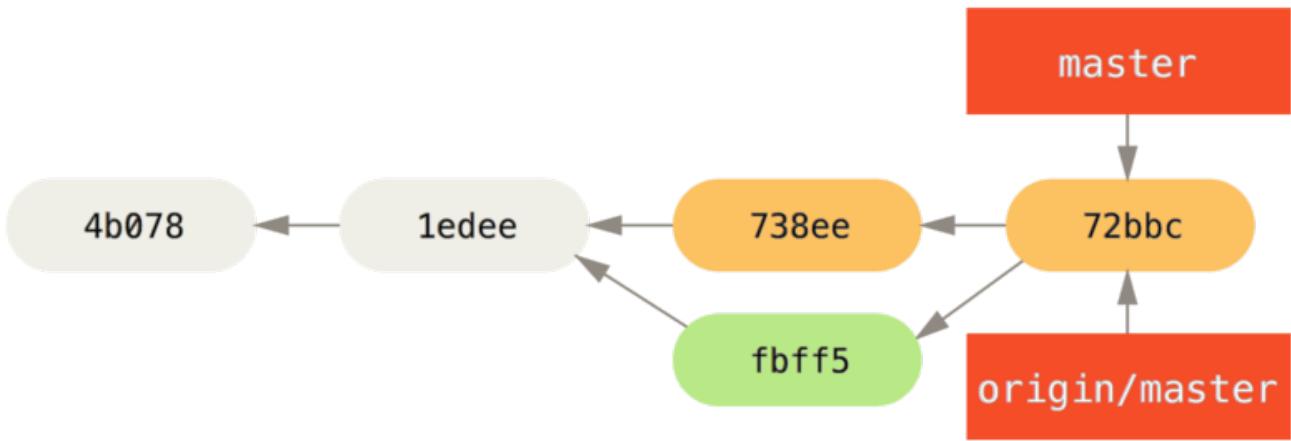


Figura 60. John's history after pushing to the `origin` server.

Mientras tanto, Jessica ha estado trabajando en una rama temática. Ella creó una rama temática llamada `issue54` y realizó tres commits en esa rama. Todavía no ha revisado los cambios de John, por lo que su historial de commit se ve así:

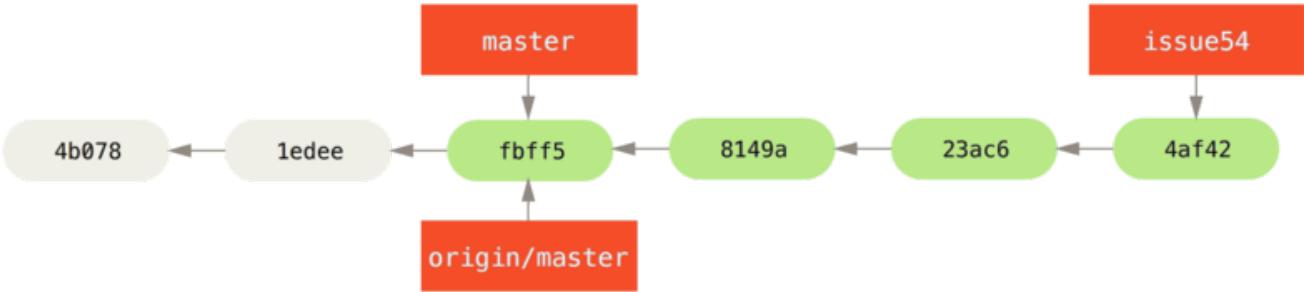


Figura 61. Jessica's topic branch.

Jessica quiere sincronizar con John, así que busca:

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

Eso reduce el trabajo que John ha impulsado mientras tanto. La historia de Jessica ahora se ve así:

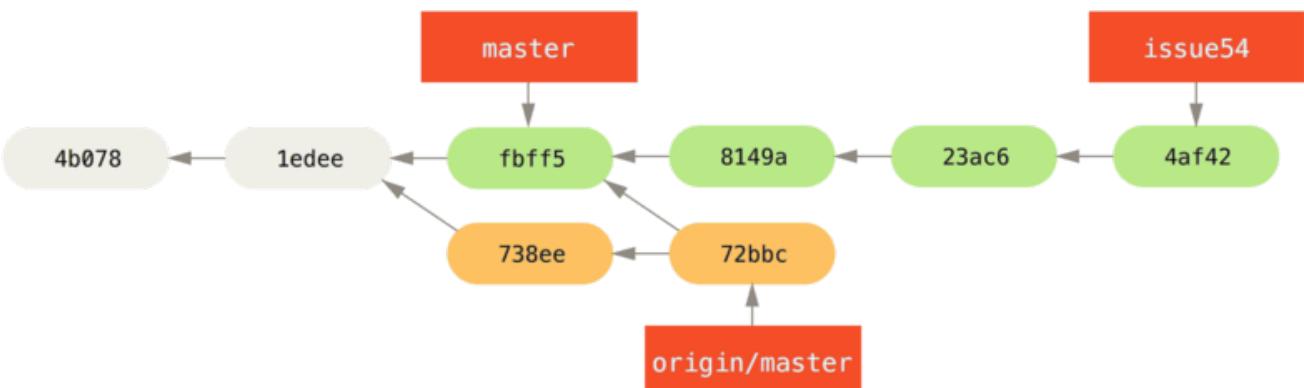


Figura 62. Jessica's history after fetching John's changes.

Jessica cree que su rama temática está lista, pero quiere saber qué tiene que fusionar en su trabajo para poder impulsarla. Ella ejecuta `git log` para descubrir:

```
$ 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
```

La sintaxis `issue54..origin / master` es un filtro de registro que le pide a Git que sólo muestre la lista de confirmaciones que están en la última rama (en este caso `origen / maestro`) que no están en la primera rama (en este caso `issue54`). Repasaremos esta sintaxis en detalle en `<<_commit_ranges>>`.

Por ahora, podemos ver en el resultado que hay un compromiso único que John ha realizado en el que Jessica no se ha fusionado. Si fusiona `origin / master`, esa es la única confirmación que modificará su trabajo local.

Ahora, Jessica puede fusionar su trabajo temático en su rama principal, fusionar el trabajo de John (`origin / master`) en su rama `master`, y luego volver al servidor nuevamente. Primero, vuelve a su rama principal para integrar todo este trabajo:

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

Ella puede fusionar ya sea `origin / master` o `issue54` primero - ambos están en sentido ascendente, por lo que el orden no importa. La instantánea final debe ser idéntica sin importar qué orden ella elija; sólo la historia será ligeramente diferente. Ella elige fusionarse en `issue54` primero:

```
$ git merge issue54
Updating fbf5bc..4af4298
Fast forward
 README      |    1 +
 lib/simplegit.rb |    6 +++++-
 2 files changed, 6 insertions(+), 1 deletions(-)
```

No hay problemas como pueden ver, fue un simple avance rápido. Ahora Jessica se fusiona en el trabajo de John (`origin / master`):

```
$ git merge origin/master
Auto-merging lib/simplegit.rb
Merge made by recursive.
 lib/simplegit.rb |    2 ++
 1 files changed, 1 insertions(+), 1 deletions(-)
```

Todo se funde limpiamente, y la historia de Jessica se ve así:

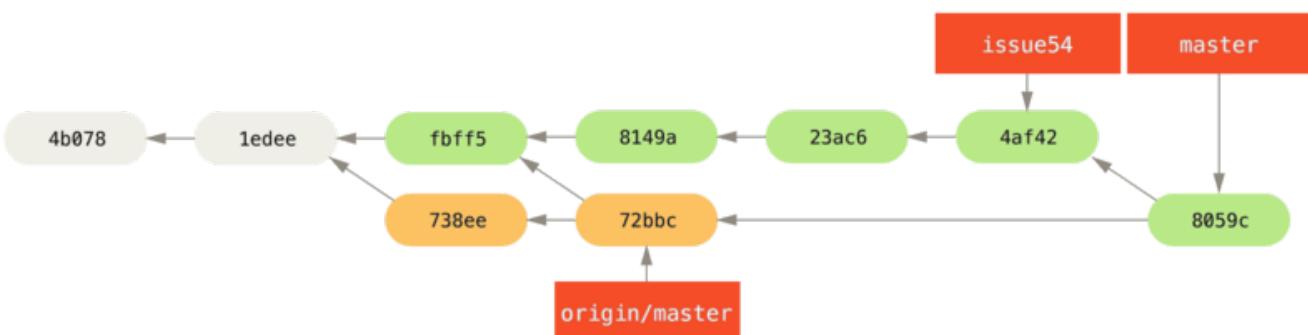


Figura 63. Jessica's history after merging John's changes.

Ahora `origin / master` es accesible desde la rama `master` de Jessica, por lo que

debería poder presionar con éxito (suponiendo que John no haya pulsado nuevamente mientras tanto):

```
$ git push origin master  
...  
To jessica@githost:simplegit.git  
 72bbc59..8059c15  master -> master
```

Cada desarrollador se ha comprometido algunas veces y se ha fusionado el trabajo de cada uno con éxito.

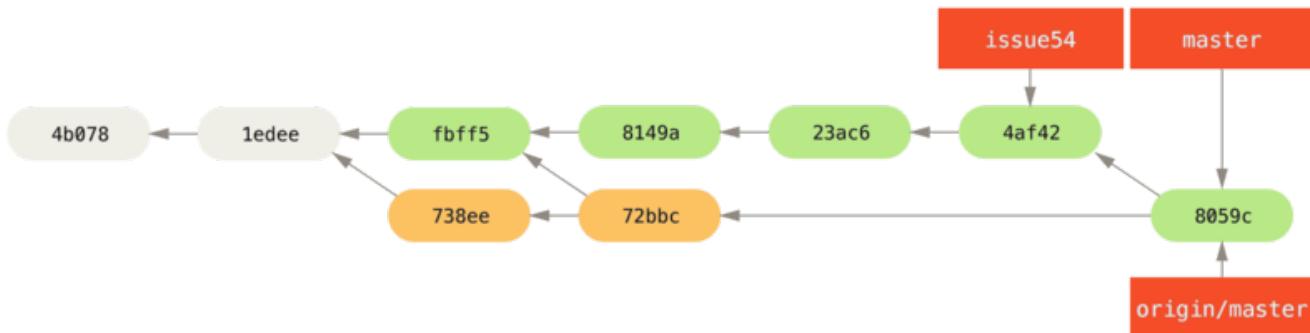


Figura 64. Jessica's history after pushing all changes back to the server.

Ese es uno de los flujos de trabajo más simples. Trabajas por un tiempo, generalmente en una rama temática, y te unes a tu rama principal cuando está lista para integrarse. Cuando desee compartir ese trabajo, hágalo en su propia rama principal, luego busque y combine `origin / master` si ha cambiado, y finalmente presione en la rama ` master` del servidor. La secuencia general es algo como esto:

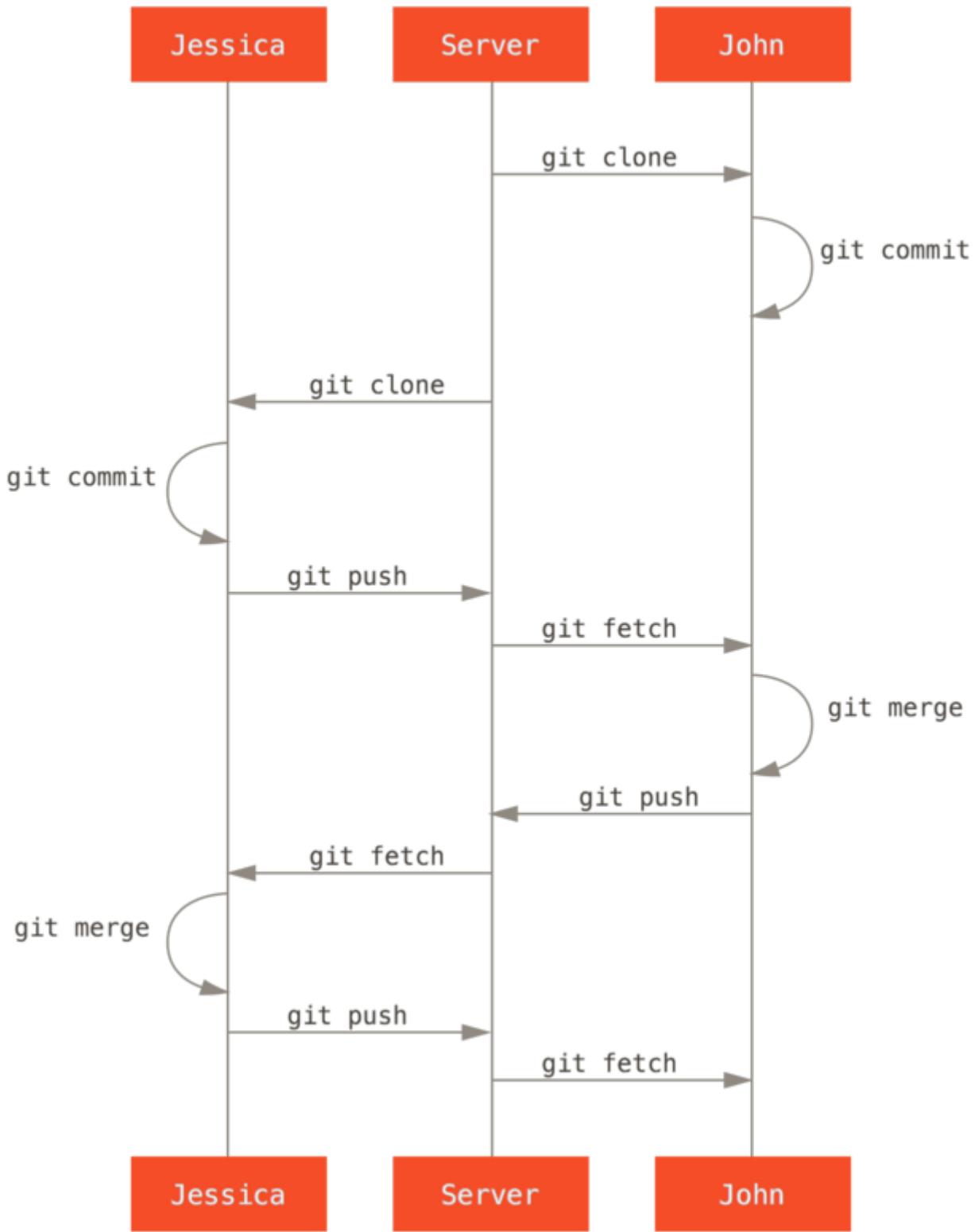


Figura 65. General sequence of events for a simple multiple-developer Git workflow.

Equipo privado administrado

En este próximo escenario, observará los roles de los contribuyentes en un grupo privado más grande. Aprenderá cómo trabajar en un entorno en el que los grupos pequeños colaboran en las funciones y luego esas contribuciones en equipo estarán integradas por otra parte.

Digamos que John y Jessica están trabajando juntos en una característica, mientras que

Jessica y Josie están trabajando en una segunda. En este caso, la compañía está utilizando un tipo de flujo de trabajo de integración-gerente donde el trabajo de los grupos individuales está integrado sólo por ciertos ingenieros, y la rama **maestra** del repositorio principal sólo puede ser actualizada por esos ingenieros. En este escenario, todo el trabajo se realiza en ramas basadas en equipos y luego los integradores lo agrupan.

Sigamos el flujo de trabajo de Jessica mientras trabaja en sus dos funciones, colaborando en paralelo con dos desarrolladores diferentes en este entorno. Suponiendo que ya haya clonado su repositorio, ella decide trabajar primero en **featureA**. Ella crea una nueva rama para la característica y hace algo de trabajo allí:

```
# 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(-)
```

En este punto, ella necesita compartir su trabajo con John, por lo que empuja a su rama **featureA** a comprometerse con el servidor. Jessica no tiene acceso de inserción a la rama **maestra**, solo los integradores lo hacen, por lo que debe enviar a otra rama para colaborar con John:

```
$ git push -u origin featureA
...
To jessica@githost:simplegit.git
 * [new branch]      featureA -> featureA
```

Jessica le envía un correo electrónico a John para decirle que ha enviado algo de trabajo a una rama llamada **featureA** y ahora puede verlo. Mientras espera los comentarios de John, Jessica decide comenzar a trabajar en **featureB** con Josie. Para comenzar, inicia una nueva rama de características, basándose en la rama **master** del servidor:

```
# Jessica's Machine
$ git fetch origin
$ git checkout -b featureB origin/master
Switched to a new branch 'featureB'
```

Ahora, Jessica hace un par de commits en la rama **featureB**:

```
$ 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(-)
```

El repositorio de Jessica se ve así:

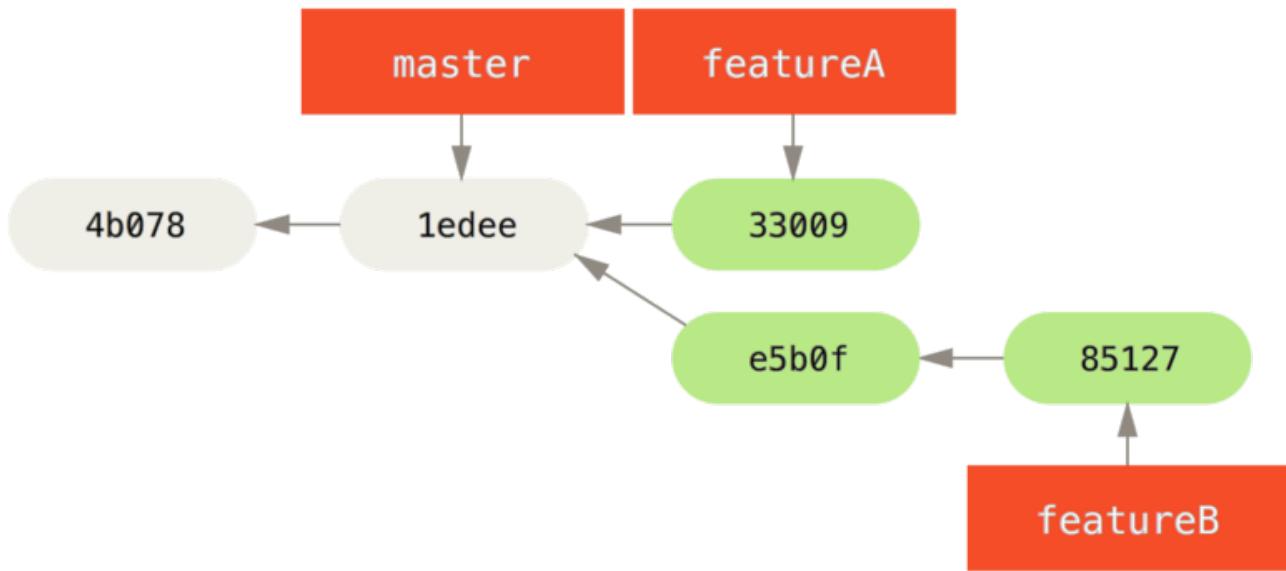


Figura 66. Jessica's initial commit history.

Está lista para impulsar su trabajo, pero recibe un correo electrónico de Josie que indica que una rama con algunos trabajos iniciales ya fue enviada al servidor como **featureBee**. Jessica primero necesita fusionar esos cambios con los suyos antes de poder presionar al servidor. Luego puede buscar los cambios de Josie con **git fetch**:

```
$ git fetch origin
...
From jessica@githost:simplegit
 * [new branch]      featureBee -> origin/featureBee
```

Jessica ahora puede fusionar esto en el trabajo que hizo con **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(-)
```

Existe un pequeño problema: necesita insertar el trabajo combinado su rama **featureB**

con la rama `featureBee` del servidor. Ella puede hacerlo especificando la rama local seguida de dos puntos (:) seguido de la rama remota al comando `git push`:

```
$ git push -u origin featureB:featureBee
...
To jessica@githost:simplegit.git
  fba9af8..cd685d1  featureB -> featureBee
```

Esto se llama *refspec*. Ver << _ refspec >> para una discusión más detallada de las referencias de Git y diferentes cosas que puedes hacer con ellas. También observe la bandera `-u`; esto es la abreviatura de `--set-upstream`, que configura las ramas para empujar y tirar más fácilmente en el futuro.

Luego, John envía un correo electrónico a Jessica para decirle que ha enviado algunos cambios a la rama `featureA` y le pide que los verifique. Ella ejecuta un `git fetch` para bajar esos cambios:

```
$ git fetch origin
...
From jessica@githost:simplegit
  3300904..aad881d  featureA -> origin/featureA
```

Luego, ella puede ver qué se ha cambiado con `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
```

Finalmente, fusiona el trabajo de John en su propia rama `featureA`:

```
$ 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 quiere modificar algo, por lo que se compromete de nuevo y luego lo envía de vuelta al servidor:

```

$ 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

```

El historial de compromiso de Jessica ahora se ve así:

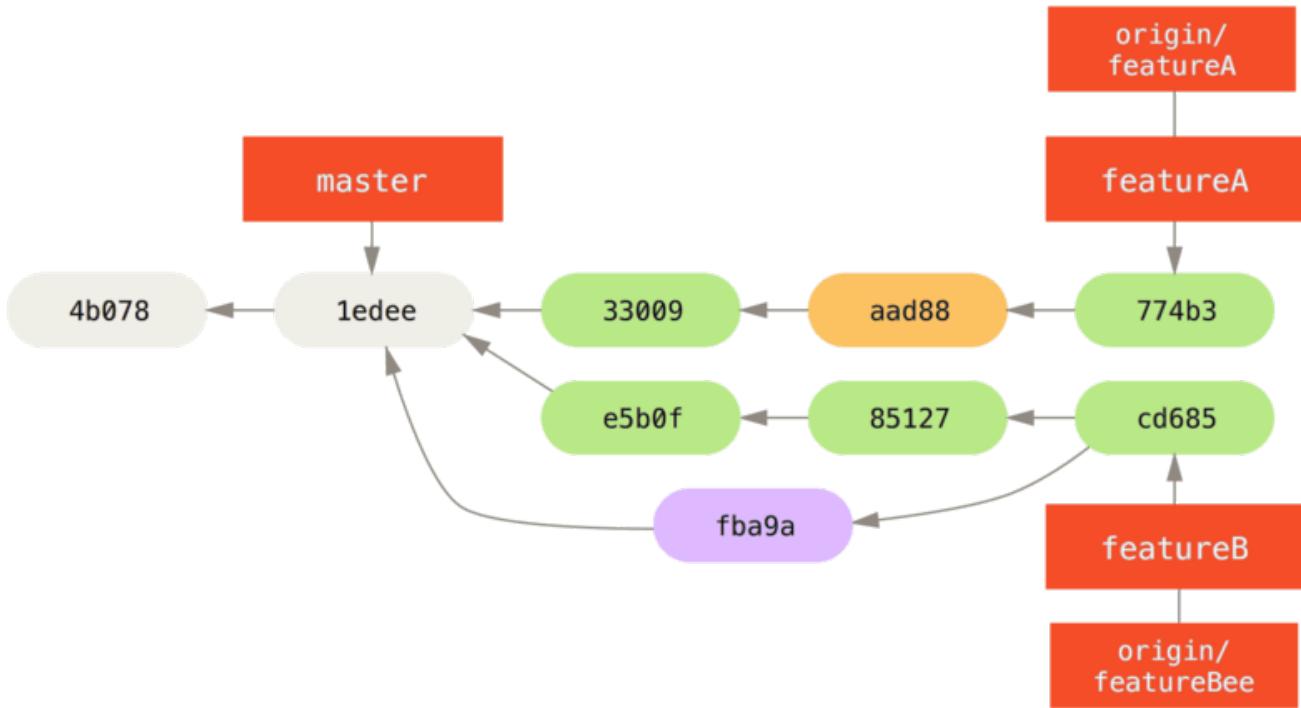


Figura 67. Jessica's history after committing on a feature branch.

Jessica, Josie y John informan a los integradores que las ramas `featureA` y `featureBee` en el servidor están listas para su integración en la línea principal. Después de que los integradores fusionen estas ramas en la línea principal, una búsqueda reducirá la nueva confirmación de fusión, haciendo que el historial se vea así:

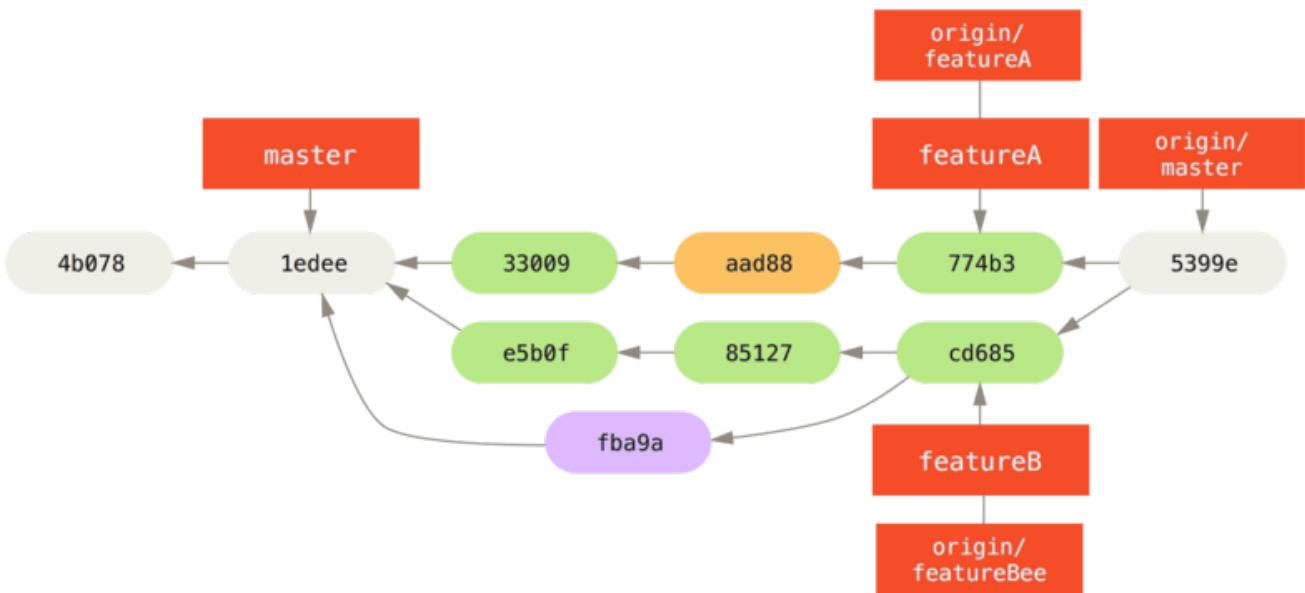


Figura 68. Jessica's history after merging both her topic branches.

Muchos grupos cambian a Git debido a esta capacidad de tener varios equipos trabajando en paralelo, fusionando las diferentes líneas de trabajo al final del proceso. La capacidad de los subgrupos más pequeños de un equipo para colaborar a través de ramas remotas, sin necesariamente tener que involucrar o impedir a todo el equipo, es un gran beneficio de Git. La secuencia del flujo de trabajo que vió aquí es algo como esto:

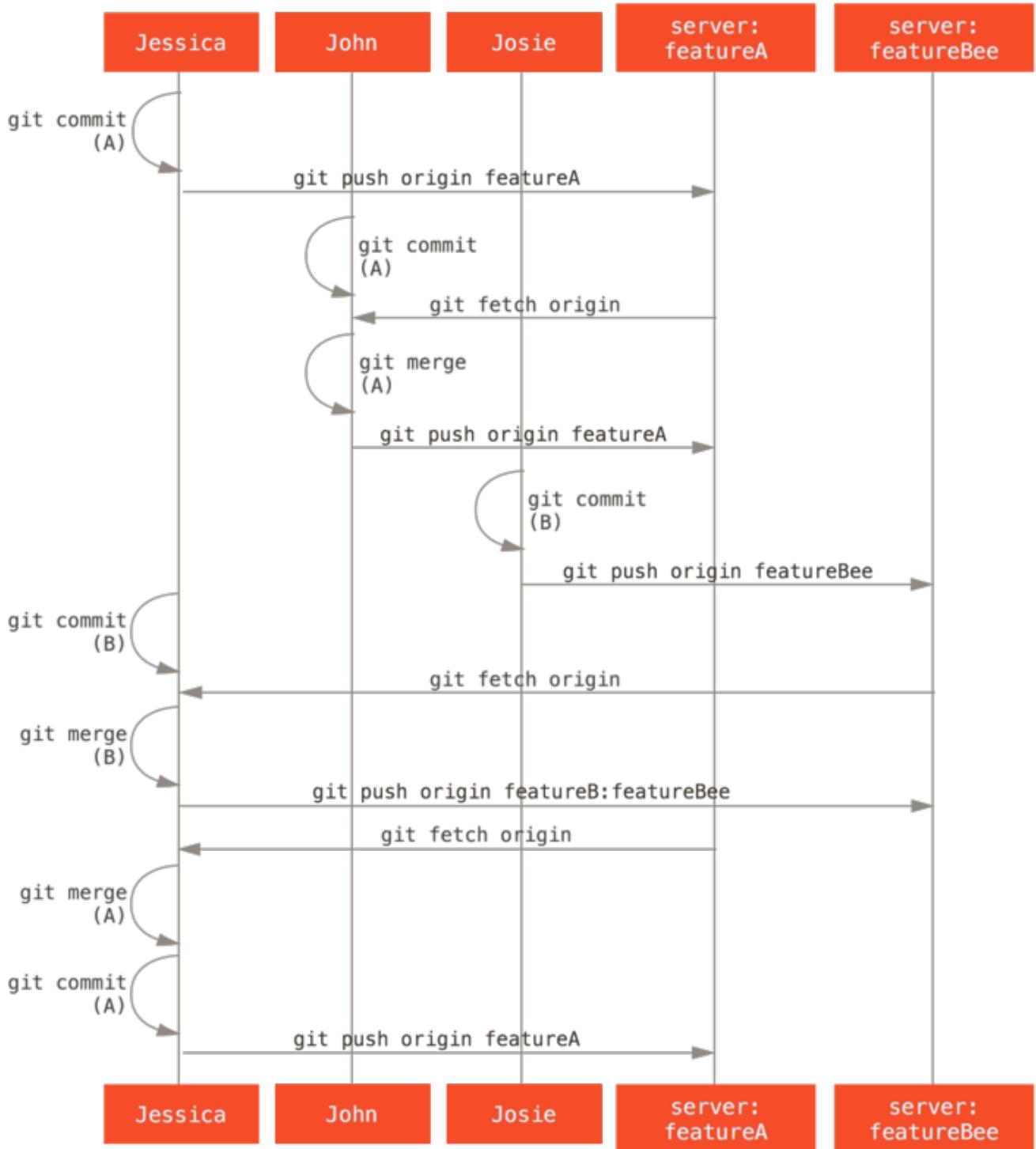


Figura 69. Basic sequence of this managed-team workflow.

Proyecto público bifurcado

Contribuir a proyectos públicos es un poco diferente. Como no tiene los permisos para actualizar directamente las ramas en el proyecto, debe obtener el trabajo de otra manera. Este primer ejemplo describe la contribución mediante bifurcación en hosts Git que admiten bifurcación fácil. Muchos sitios de alojamiento admiten esto (incluidos GitHub, BitBucket, Google Code, repo.or.cz entre otros), y muchos mantenedores de proyectos esperan este estilo de contribución. La siguiente sección trata de proyectos que prefieren aceptar parches contribuidos por correo electrónico.

En primer lugar, es probable que desee clonar el repositorio principal, crear una rama

de tema para el parche o la serie de parches en que planea contribuir y hacer su trabajo allí. La secuencia se ve básicamente así:

```
$ git clone (url)
$ cd project
$ git checkout -b featureA
# (work)
$ git commit
# (work)
$ git commit
```

NOTA

Puede usar `rebase -i` para reducir su trabajo a una única confirmación, o reorganizar el trabajo en las confirmaciones para que el desarrollador pueda revisar el parche más fácilmente. Consulte << _ rewriting_history >> para obtener más información sobre el rebase interactivo.

Cuando finalice el trabajo en la rama y esté listo para contribuir con los mantenedores, vaya a la página original del proyecto y haga clic en el botón “ Tenedor ”, creando su propio tenedor escribible del proyecto. Luego debe agregar esta nueva URL de repositorio como segundo control remoto, en este caso llamado `myfork`:

```
$ git remote add myfork (url)
```

Entonces necesitas impulsar tu trabajo hasta esa nueva URL. Es más fácil impulsar la rama de tema en la que está trabajando hasta su repositorio, en lugar de fusionarse con su rama principal y aumentarla. La razón es que si el trabajo no se acepta o se selecciona con una cereza, no es necesario rebobinar la rama maestra. Si los mantenedores se fusionan, redimensionan o seleccionan su trabajo, eventualmente lo recuperará a través de su repositorio de todas maneras:

```
$ git push -u myfork featureA
```

Cuando su trabajo ha sido empujado hacia su tenedor, debe notificar al mantenedor. Esto a menudo se denomina solicitud de extracción, y puede generarlo a través del sitio web: GitHub tiene su propio mecanismo de solicitud de extracción que veremos en << _ github >> o puede ejecutar el comando `git request-pull` y enviar por correo electrónico la salida al mantenedor del proyecto de forma manual.

El comando `request-pull` toma la rama base en la que desea que se saque su rama de tema y la URL del repositorio de Git de la que desea que extraigan, y genera un resumen de todos los cambios que está solicitando. Por ejemplo, si Jessica quiere enviar a John una solicitud de extracción, y ella ha hecho dos commits en la rama de temas que acaba de subir, puede ejecutar esto:

```
$ 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(-)
```

La salida se puede enviar al mantenedor, le dice de dónde se ramificó el trabajo, resume los compromisos y dice de dónde sacar este trabajo.

En un proyecto para el cual no eres el mantenedor, generalmente es más fácil tener una rama como `master` siempre rastreando `origin / master` y hacer tu trabajo en ramas de tema que puedes descartar fácilmente si son rechazadas. Tener temas de trabajo aislados en las ramas temáticas también facilita la tarea de volver a establecer una base de trabajo si la punta del repositorio principal se ha movido mientras tanto y sus confirmaciones ya no se aplican limpiamente. Por ejemplo, si desea enviar un segundo tema de trabajo al proyecto, no continúe trabajando en la rama de tema que acaba de crear: vuelva a comenzar desde la rama `master` del repositorio principal:

```
$ git checkout -b featureB origin/master
# (work)
$ git commit
$ git push myfork featureB
# (email maintainer)
$ git fetch origin
```

Ahora, cada uno de sus temas está contenido dentro de un silo, similar a una fila de parches, que puede volver a escribir, volver a establecer y modificar sin que los temas interfieran o se interrelacionen entre sí, de la siguiente manera:

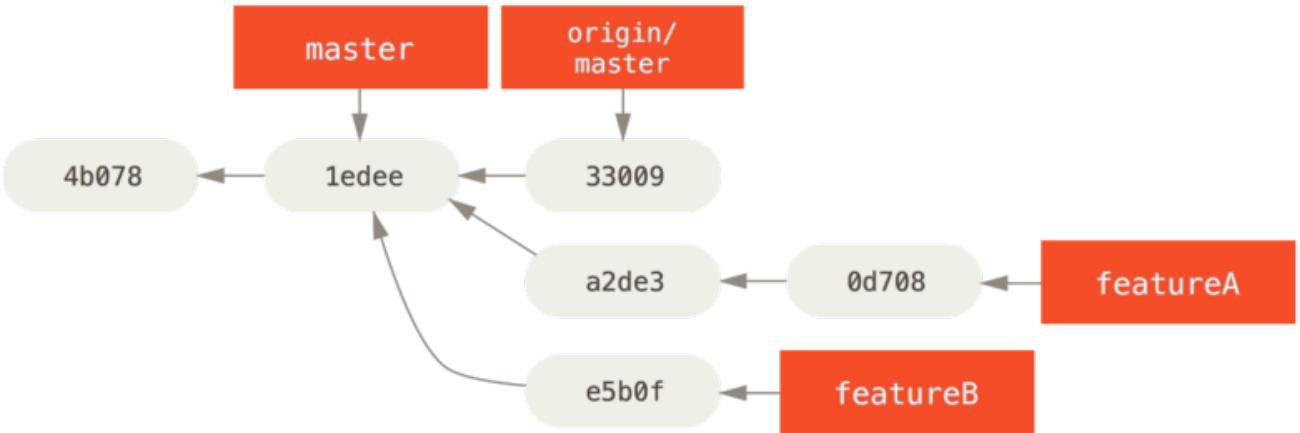


Figura 70. Initial commit history with **featureB** work.

Digamos que el mantenedor del proyecto ha sacado otros parches y ha probado su primera rama, pero ya no se fusiona de manera limpia. En este caso, puede tratar de volver a establecer la base de esa rama sobre *origin / master*, resolver los conflictos del mantenedor y luego volver a enviar los cambios:

```
$ git checkout featureA
$ git rebase origin/master
$ git push -f myfork featureA
```

Esto reescribe tu historial para que ahora parezca << psp_b >>.

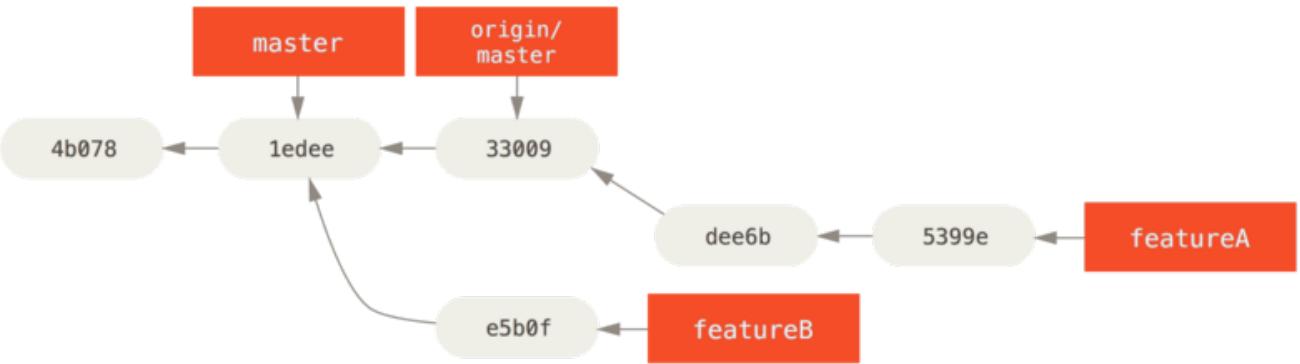


Figura 71. Commit history after **featureA** work.

Debido a que rebasaste la rama, debes especificar el **-f** en tu comando push para poder reemplazar la rama `featureA` en el servidor con una confirmación que no sea un descendiente de ella. Una alternativa sería llevar este nuevo trabajo a una rama diferente en el servidor (tal vez llamada **featureAv2**).

Veamos un escenario más posible: el mantenedor ha observado el trabajo en su segunda rama y le gusta el concepto, pero le gustaría que cambie un detalle de implementación. También aprovechará esta oportunidad para mover el trabajo basado en la rama **maestra** 'actual del proyecto. Usted inicia una nueva rama basada en la rama actual de 'origen / maestro', aplasta los cambios 'featureB' allí, resuelve cualquier conflicto, hace que la implementación cambie, y luego lo empuja hacia arriba como una nueva rama:

```
$ git checkout -b featureBv2 origin/master
$ git merge --no-commit --squash featureB
# (change implementation)
$ git commit
$ git push myfork featureBv2
```

La opción `--squash` toma todo el trabajo en la rama fusionada y lo aplasta en una confirmación sin fusión en la parte superior de la rama en la que se encuentra. La opción `--no-commit` le dice a Git que no registre automáticamente una confirmación. Esto le permite introducir todos los cambios desde otra rama y luego realizar más cambios antes de registrar la nueva confirmación.

Ahora puede enviar al mantenedor un mensaje de que ha realizado los cambios solicitados y puede encontrar esos cambios en su rama `featureBv2`.

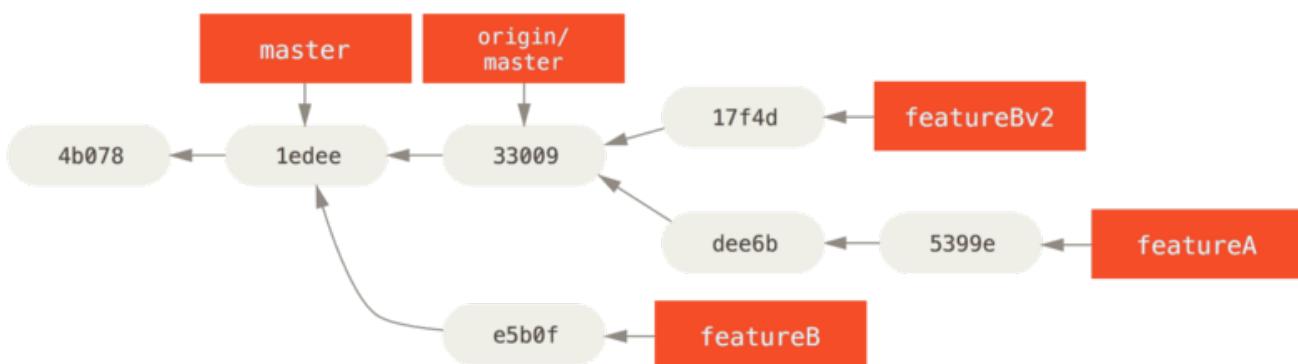


Figura 72. Commit history after `featureBv2` work.

Proyecto público a través de correo electrónico

Muchos proyectos han establecido procedimientos para aceptar parches: deberá verificar las reglas específicas para cada proyecto, ya que serán diferentes. Dado que hay varios proyectos antiguos y más grandes que aceptan parches a través de una lista de correo electrónico para desarrolladores, veremos un ejemplo de eso ahora.

El flujo de trabajo es similar al caso de uso anterior: crea ramas de tema para cada serie de parches en las que trabaja. La diferencia es cómo los envía al proyecto. En lugar de bifurcar el proyecto y avanzar hacia su propia versión de escritura, genera versiones de correo electrónico de cada serie de commit y las envía por correo electrónico a la lista de correo de desarrolladores:

```
$ git checkout -b topicA
# (work)
$ git commit
# (work)
$ git commit
```

Ahora tiene dos confirmaciones que desea enviar a la lista de correo. Utiliza `git format-patch` para generar los archivos con formato mbox que puedes enviar por correo electrónico a la lista: convierte cada confirmación en un mensaje de correo electrónico con la primera línea del mensaje de confirmación como tema y el resto de el mensaje más el parche que introduce el compromiso como el cuerpo. Lo bueno de esto es que la aplicación de un parche de un correo electrónico generado con `format-patch` conserva toda la información de compromiso correctamente.

```
$ git format-patch -M origin/master  
0001-add-limit-to-log-function.patch  
0002-changed-log-output-to-30-from-25.patch
```

El comando `format-patch` imprime los nombres de los archivos de parche que crea. El modificador `-M` le dice a Git que busque cambios de nombre. Los archivos terminan pareciéndose a esto:

```
$ 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
```

También puede editar estos archivos de parche para agregar más información para la lista de correo electrónico que no desea mostrar en el mensaje de confirmación. Si agrega texto entre la línea `---` y el comienzo del parche (la línea `diff - git`), los desarrolladores pueden leerlo; pero aplicar el parche lo excluye.

Para enviarlo por correo electrónico a una lista de correo, puede pegar el archivo en su programa de correo electrónico o enviarlo a través de un programa de línea de comandos. Pegar el texto a menudo causa problemas de formateo, especialmente con clientes “más inteligentes” que no conservan líneas nuevas y otros espacios en blanco de manera apropiada. Afortunadamente, Git proporciona una herramienta para ayudarlo a enviar parches con formato correcto a través de IMAP, que puede ser más fácil para usted. Demostraremos cómo enviar un parche a través de Gmail, que es el agente de correo electrónico que mejor conocemos; puede leer instrucciones detalladas para una cantidad de programas de correo al final del archivo [Documentation / SubmittingPatches](#) antes mencionado en el código fuente de Git.

Primero, necesitas configurar la sección imap en tu archivo `~/.gitconfig`. Puede establecer cada valor por separado con una serie de comandos `git config`, o puede agregarlos manualmente, pero al final su archivo de configuración debería verse más o menos así:

```
[imap]
  folder = "[Gmail]/Drafts"
  host = imaps://imap.gmail.com
  user = user@gmail.com
  pass = p4ssw0rd
  port = 993
  sslverify = false
```

Si su servidor IMAP no usa SSL, las dos últimas líneas probablemente no sean necesarias, y el valor del host será `imap: //` en lugar de `imaps: //`. Cuando esté configurado, puede usar `git send-email` para colocar la serie de parches en la carpeta Borradores del servidor IMAP especificado:

```
$ 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
```

Entonces, Git escupe un montón de información de registro que se ve algo así para cada parche que está enviando:

```
(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

En este punto, debe poder ir a la carpeta Borradores, cambiar el campo “A”(To) a la lista de correo a la que le envía el parche, posiblemente CC al responsable de esa sección y enviarlo.

Resumen

Esta sección ha cubierto una cantidad de flujos de trabajo comunes para tratar con varios tipos muy diferentes de proyectos de Git que probablemente encuentre, e introdujo algunas herramientas nuevas para ayudarlo a administrar este proceso. A continuación, verá cómo trabajar la otra cara de la moneda: mantener un proyecto de Git. Aprenderá cómo ser un dictador benevolente o un gerente de integración.

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](#) 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 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 este formato – 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 nueva 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 archivos 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 archivos 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, puesto 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 antiguos y cosas similares.

Para aplicar un parche generado con `format-patch`, usa `git am`. Técnicamente, `git am` se construyó para leer de un archivo mbox (buzón de correo). Es un formato de texto plano simple para almacenar uno o más mensajes de correo en un archivo 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 archivo 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 archivo 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 archivo 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 archivo 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 fué 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 archivo para el cual detecte problemas, como si fuera una operación **merge** o **rebase**. Estos problemas se solucionan de la misma forma - edita el archivo para resolver el conflicto, prepara (stage) el nuevo archivo 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 quieres 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`, aparecería un conflicto.

Si estás aplicando varios parches a partir de un archivo mbox, también puedes ejecutar el comando `am` en modo interactivo, el cual se detiene en cada parche para preguntar si quieres 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 e-mail 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 e-mail 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 el trabajo 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 integración (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 generará a partir de un commit público al que tengas acceso.

Si no trabajas regularmente con alguien pero aún así quieres 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 quieres hacer con ella. Esta sección repasa un par de comandos 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 archivo 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 históricos 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 [Historial con varias ramas puntuales](#), e integramos primero `ruby_client` y luego `php_client`, entonces tu historial terminará con este aspecto [Tras integrar una rama puntual..](#)

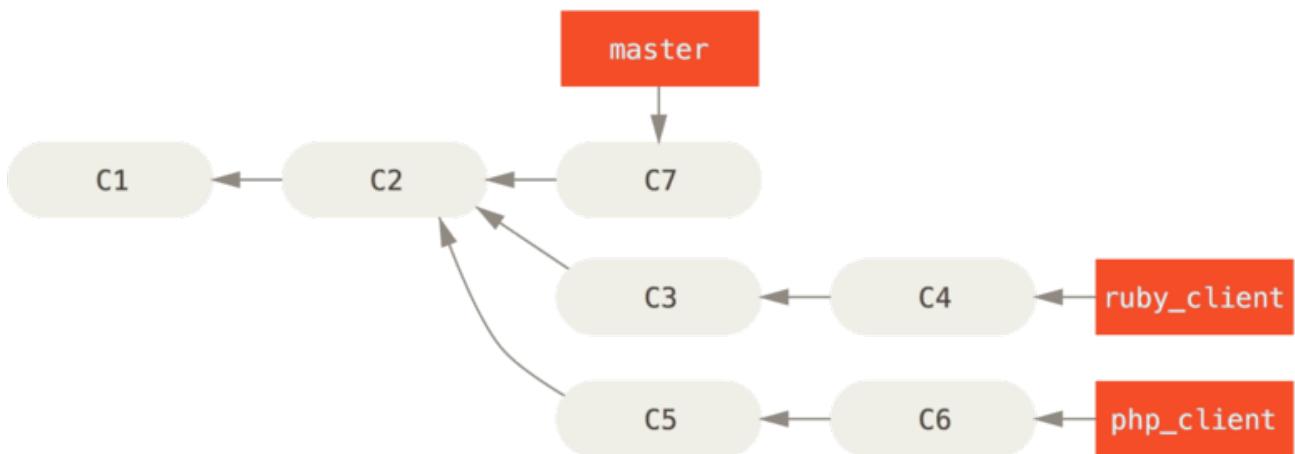


Figura 73. Historial con varias ramas puntuales.

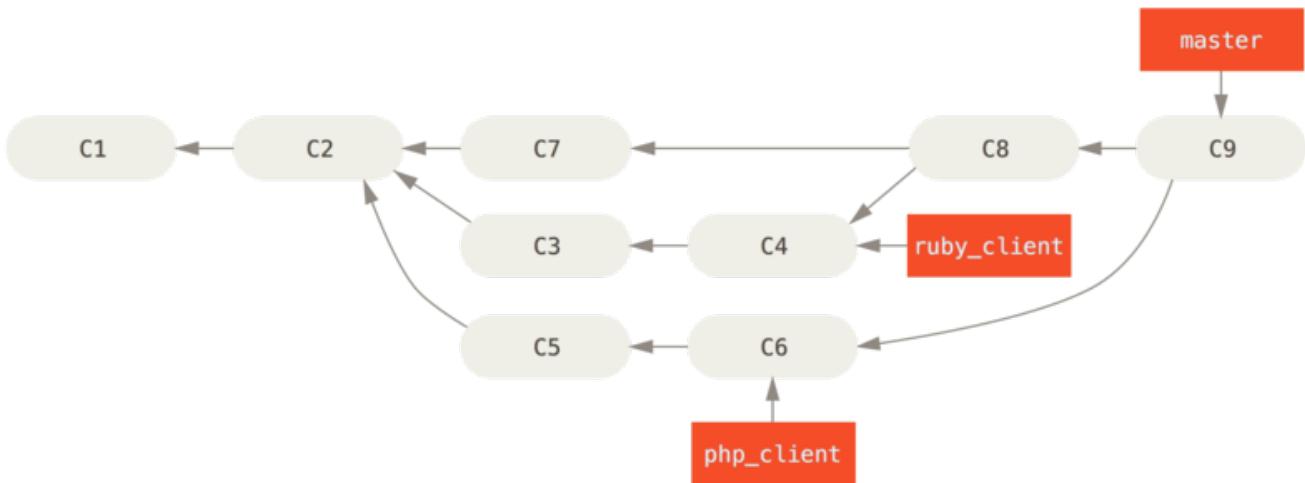


Figura 74. Tras integrar una rama puntual.

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 estables, 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 se envían habitualmente al repositorio público. Cada vez que tengas una nueva rama puntual para integrar en ([Antes de integrar una rama puntual](#)), primero la fusionas con la rama `develop` ([Tras integrar una rama puntual](#)); luego, tras etiquetar la versión, avanzas la rama `master` hasta el punto donde se encuentre la ahora estable rama `develop` ([Tras el lanzamiento de una rama puntual](#)).

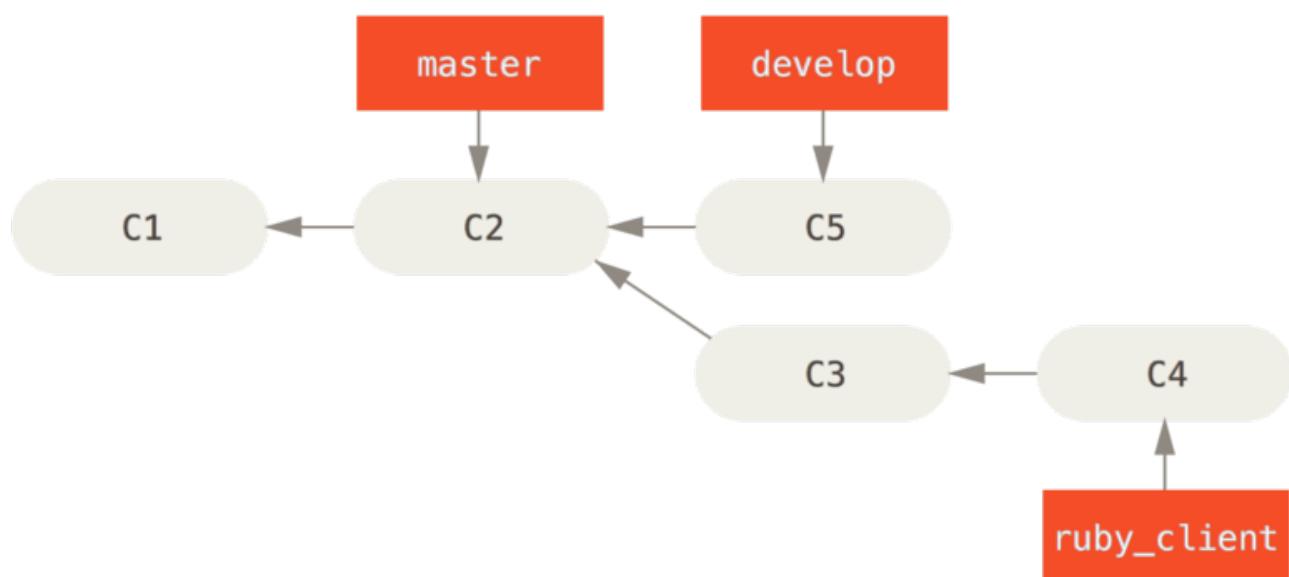


Figura 75. Antes de integrar una rama puntual.

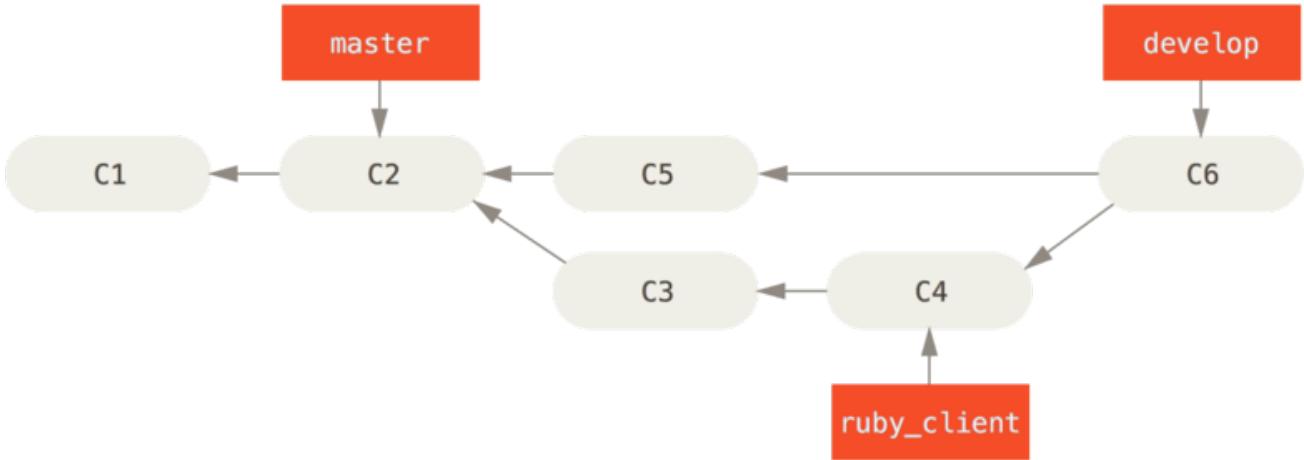


Figura 76. Tras integrar una rama puntual.

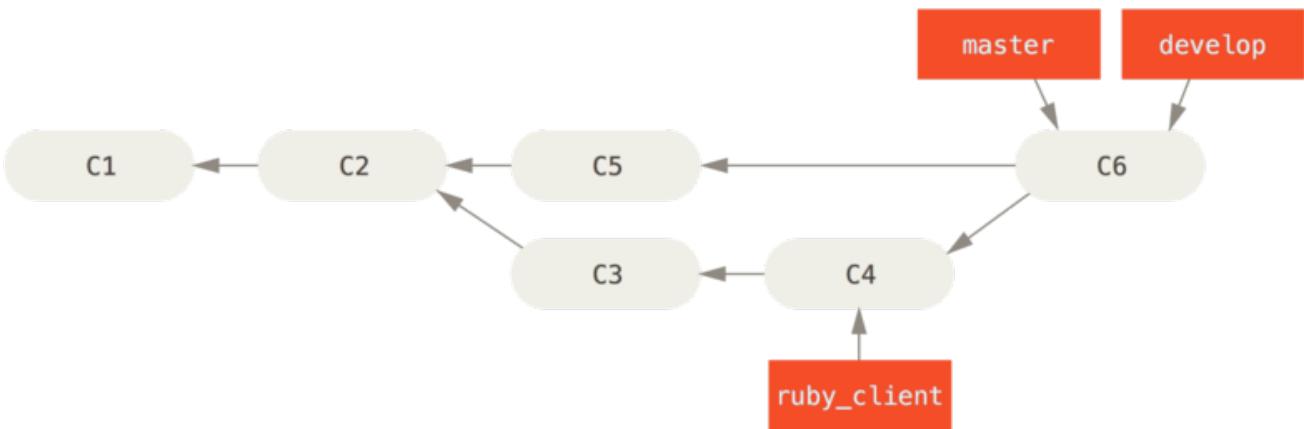


Figura 77. Tras el lanzamiento de una rama puntual.

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 [Gestionando un conjunto complejo de ramas puntuales paralelas](#)). 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.

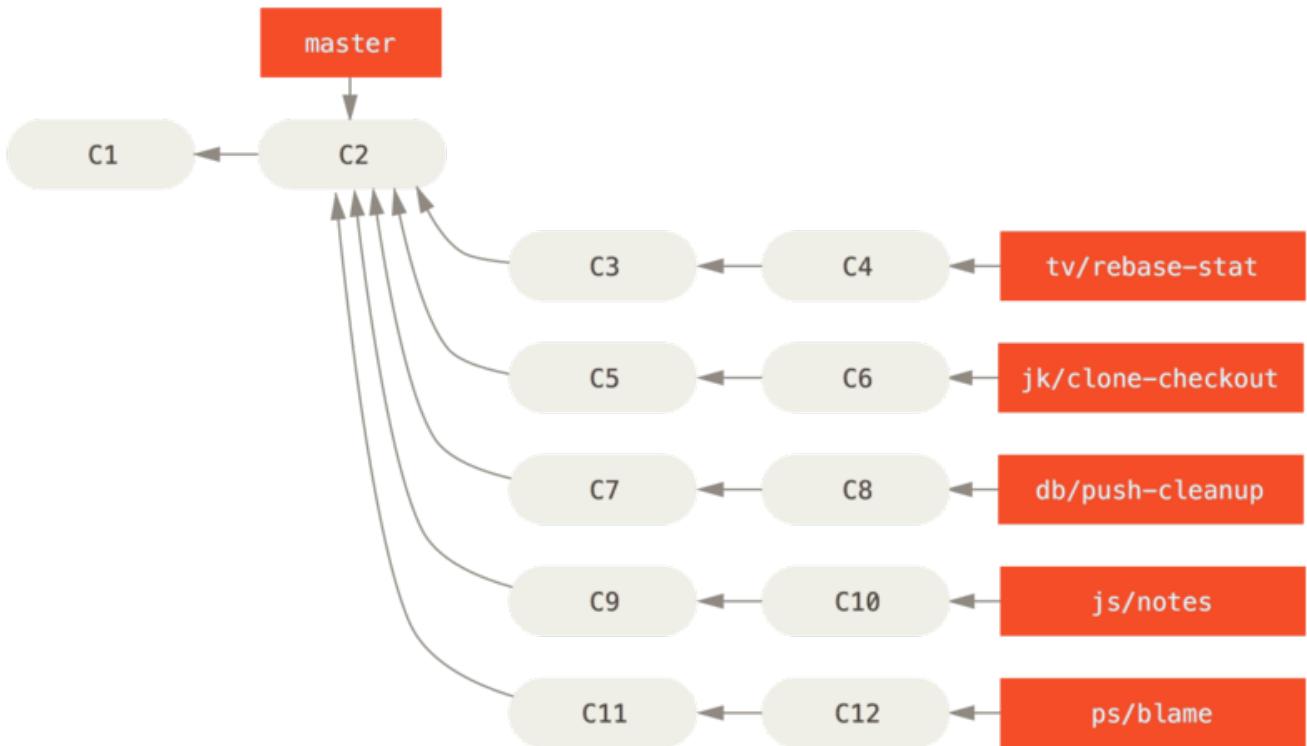


Figura 78. Gestionando un conjunto complejo de ramas puntuales paralelas.

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:

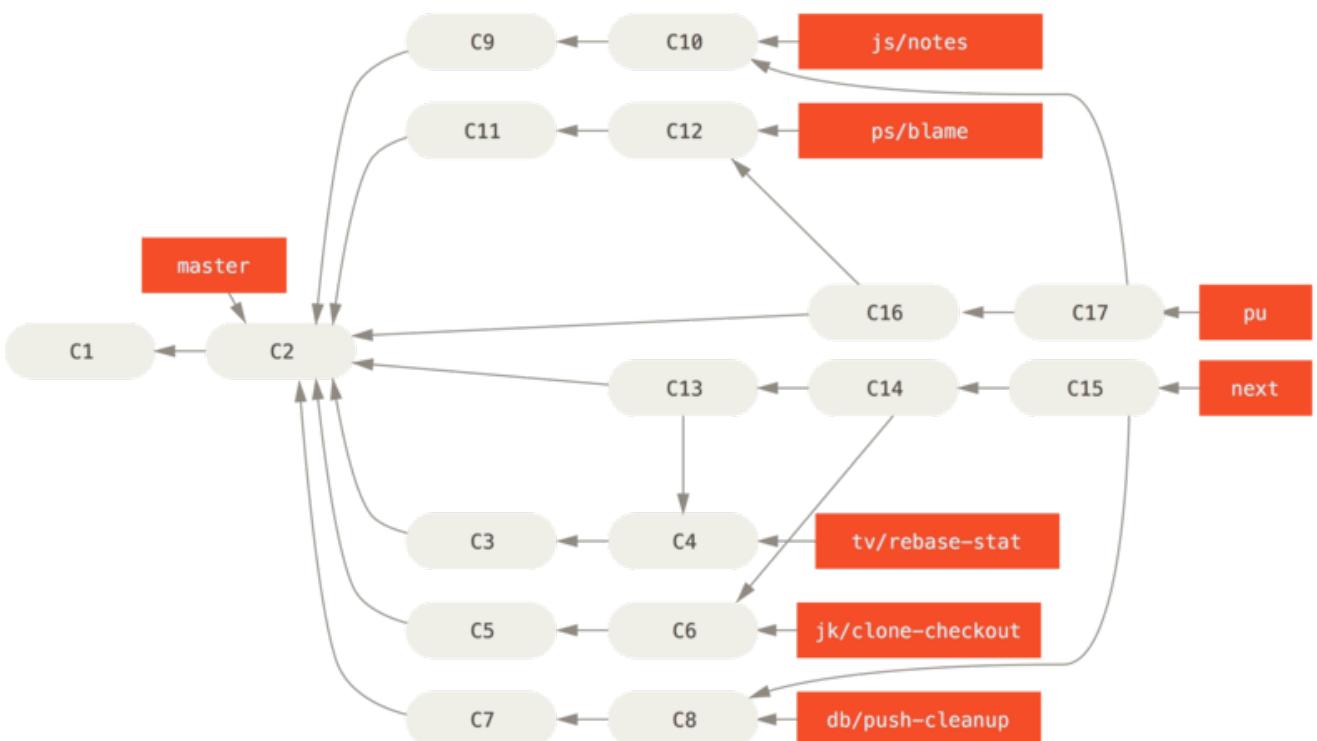


Figura 79. Fusionando ramas puntuales en ramas de integración de largo recorrido.

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 quieras 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 quieras 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:

Figura 80. 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:

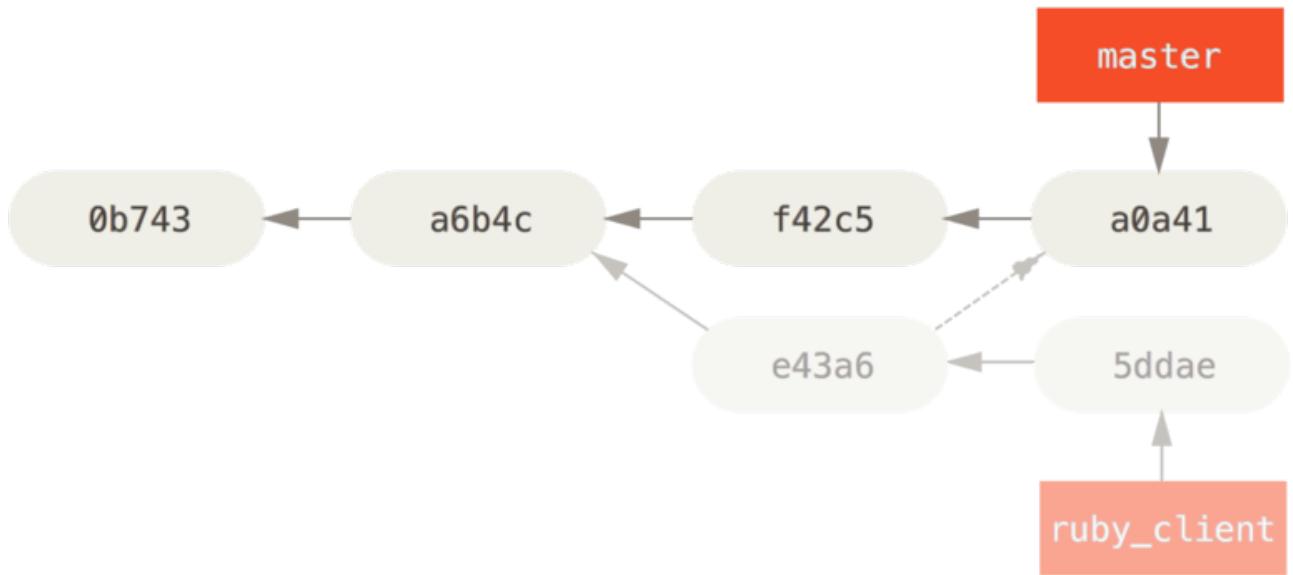


Figura 81. Historial tras entresacar un commit de una rama puntual.

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 a 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 caché 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 [Fundamentos de Git](#). 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 quieras 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 para 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 archivo 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

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 [Herramientas de Git](#).

Cambios en la interfaz

AVISO

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”.

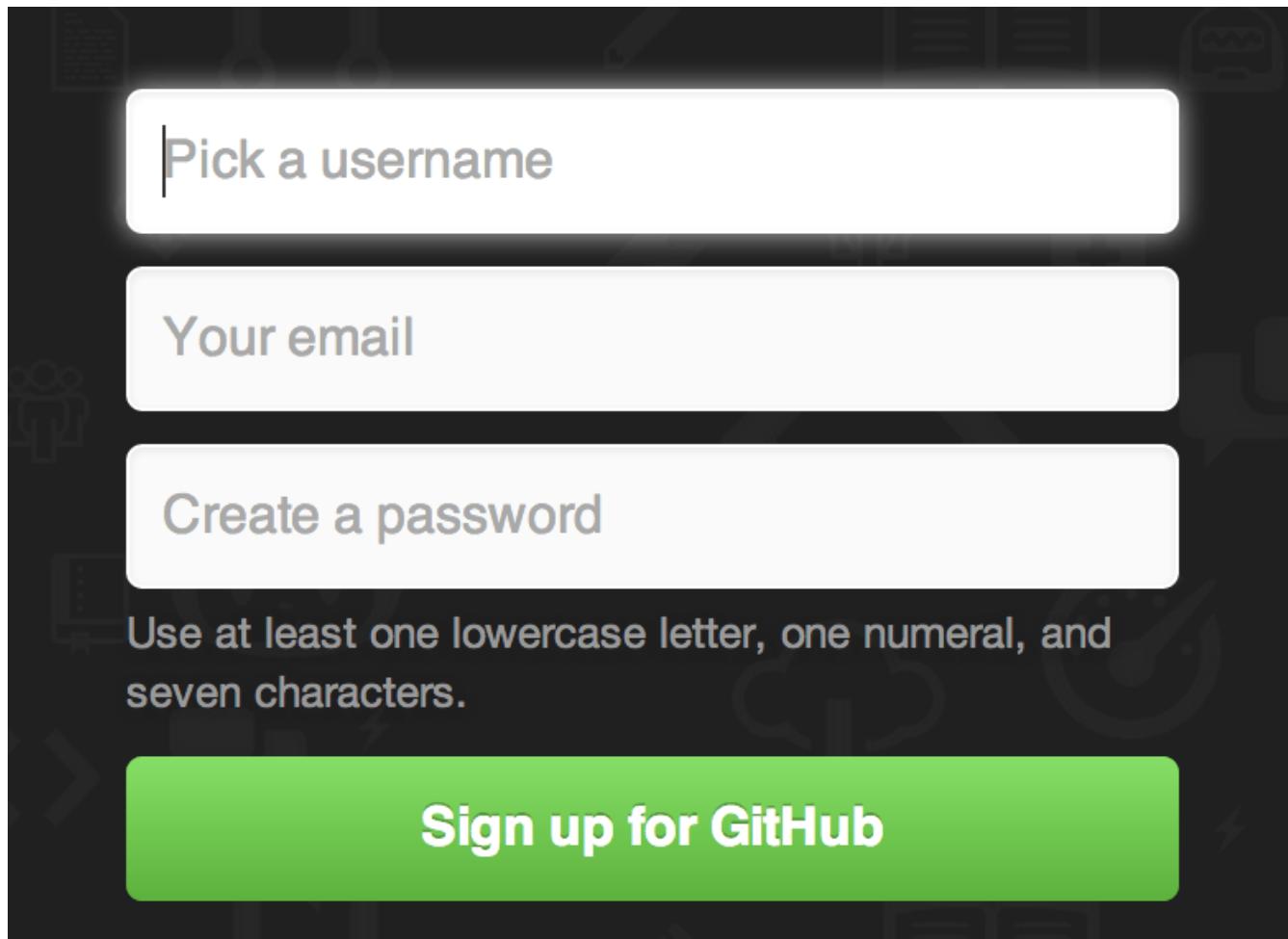


Figura 82. Formulario para darse de alta en GitHub.

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. Confirmar la dirección ahora, es bastante importante (como veremos después).

NOTA

GitHub proporciona toda su funcionalidad en cuentas gratuitas, puedes tener tanto proyectos públicos como privados ilimitados. La única limitación ese que en cada uno de tus proyectos privados solo puedes tener un máximo de tres colaboradores. Los planes de pago de GitHub te permiten tener algunas herramientas extra, 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 sólo 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 aún 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:

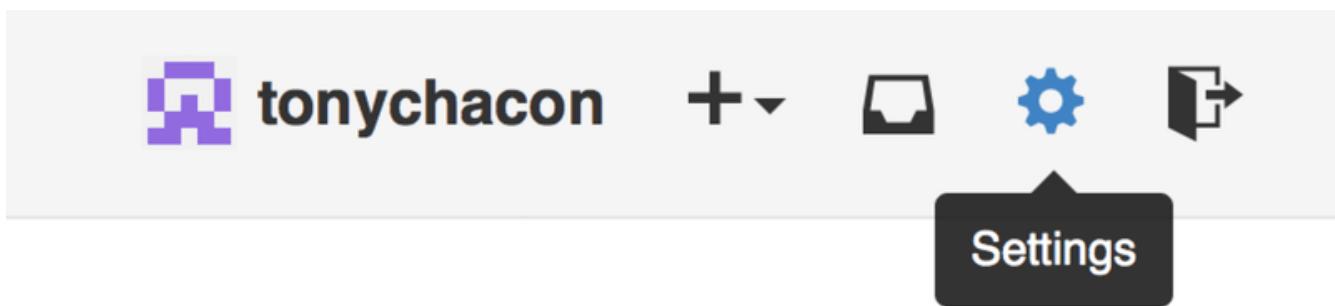


Figura 83. Enlace “Account settings”.

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, there's a sidebar with the same navigation options as Figure 83. The "SSH keys" option is selected. The main area has a heading "SSH Keys" with a "Need help? Check out our guide to [generating SSH keys](#) or troubleshoot [common SSH Problems](#)" message. Below it is a message "There are no SSH keys with access to your account." A "Add SSH key" button is located in the top right of this section. Below this is a form titled "Add an SSH Key" with fields for "Title" (an input field) and "Key" (a large text area). At the bottom of the form is a green "Add key" button.

Figura 84. Enlace “SSH keys”.

Desde ahí, pulsa sobre "[Add an SSH key](#)", proporcionando un nombre y pegando los contenidos del archivo `~/.ssh/id_rsa.pub` (o donde hayas definido tu clave pública) en el área de texto, y pulsa sobre “Add key”.

NOTA

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 saber cuál es.

Tu icono

También, si quieres, puedes reemplazar el icono (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”.

The screenshot shows the GitHub profile settings interface. On the left, a sidebar lists options: Profile (selected), Account settings, Emails, Notification center, Billing, SSH keys, Security, Applications, Repositories, and Organizations. The main area is titled "Public profile". It includes fields for "Profile picture" (with a placeholder image and "Upload new picture" button), "Name" (empty input field), "Email (will be public)" (empty input field), "URL" (empty input field), "Company" (empty input field), and "Location" (empty input field). At the bottom is a green "Update profile" button.

Figura 85. Enlace “Profile”.

Nosotros elegiremos como ejemplo una copia del logo de Git que tengamos en el disco duro y luego tendremos la opción de recortarlo al subirlo.

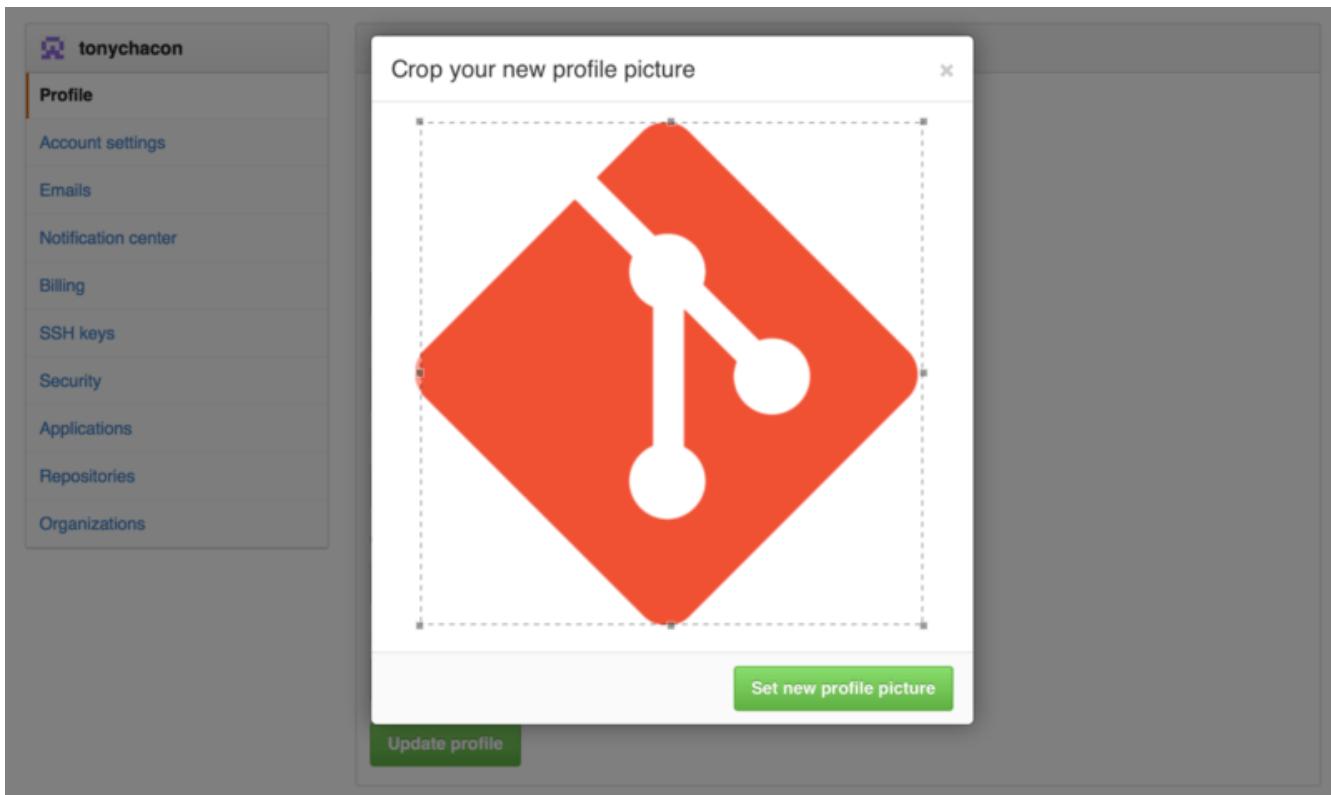


Figura 86. Recortar tu ícono

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 este paso, si no lo deseas.

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 el apartado Emails de la sección de administración.

Your **primary GitHub email address** will be used for account-related notifications (e.g. account changes and billing receipts) as well as any web-based GitHub operations (e.g. edits and merges).

| | | | |
|-------------------------|------------|--------|--|
| tonychacon@example.com | Primary | Public | Delete |
| tchacon@example.com | | | Set as primary Delete |
| tony.chacon@example.com | Unverified | | Send verification email Delete |

Add email address Add

Keep my email address private
We will use **tonychacon@users.noreply.github.com** when performing Git operations and sending email on your behalf.

Figura 87. Añadiendo direcciones de correo

En [Añadiendo direcciones de correo](#) 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 manera 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.

The screenshot shows the GitHub 'Security' settings page for the user 'tonychacon'. On the left, a sidebar lists account management options: Profile, Account settings, Emails, Notification center, Billing, SSH keys, Security (which is selected), Applications, Repositories, and Organizations. The main content area has two sections: 'Two-factor authentication' and 'Sessions'. In 'Two-factor authentication', the status is 'Off' with a red 'X'. A button labeled 'Set up two-factor authentication' is present. A note explains that two-factor authentication provides another layer of security. In the 'Sessions' section, it lists a single active session: 'Paris 85.168.227.34' (Your current session), which is a 'Safari on OS X 10.9.4' browser from 'Paris, Ile-de-France, France' signed in on 'September 30, 2014'.

Figura 88. 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 autentificación, 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 almacenará en tu cuenta y podrás escribir en él sin limitaciones.

NOTA

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.



Figura 89. 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 [Ramificaciones en Git](#).

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 [Flujo de Trabajo Administrador-Integración](#), 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 a GitHub repository page for 'schacon / blink'. At the top, there are buttons for 'Watch 0', 'Star 0', and 'Fork 0'. Below the header, it says 'branch: master' and 'blink / blink.ino'. A profile picture of schacon is shown next to the text 'schacon on Jun 12 my arduino blinking code (from arduino.cc)'. It indicates '1 contributor'. The main area displays the 'blink.ino' code with 25 lines (20 sloc) and a size of 0.71 kb. The code is as follows:

```
1  /*
2   * Blink
3   * Turns on an LED on for one second, then off for one second, repeatedly.
4   *
5   * This example code is in the public domain.
6   */
7
8 // Pin 13 has an LED connected on most Arduino boards.
9 // give it a name:
10 int led = 13;
11
12 // the setup routine runs once when you press reset:
13 void setup() {
14     // initialize the digital pin as an output.
15     pinMode(led, OUTPUT);
16 }
17
18 // the Loop routine runs over and over again forever:
19 void loop() {
20     digitalWrite(led, HIGH);    // turn the LED on (HIGH is the voltage level)
21     delay(1000);              // wait for a second
22     digitalWrite(led, LOW);    // turn the LED off by making the voltage LOW
23     delay(1000);              // wait for a second
24 }
```

Figura 90. 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 ①
Cloning into 'blink'...

$ cd blink
$ git checkout -b slow-blink ②
Switched to a new branch 'slow-blink'

$ sed -i '' 's/1000/3000/' blink.ino ③

$ git diff --word-diff ④
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' ⑤
[slow-blink 5ca509d] three seconds is better
 1 file changed, 2 insertions(+), 2 deletions(-)

$ git push origin slow-blink ⑥
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.

Example file to blink the LED on an Arduino — Edit

2 commits 2 branches 0 releases 1 contributor

Your recently pushed branches:

slow-blink (less than a minute ago) Compare & pull request

This branch is even with schacon:master

Create README.md

schacon authored on Jun 12 latest commit bbc80f9b29

README.md Create README.md 4 months ago

blink.ino my arduino blinking code (from arduino.cc) 4 months ago

README.md

Blink

This repository has an example file to blink the LED on an Arduino board.

Code Pull Requests Wiki Pulse Graphs Settings

HTTPS clone URL
https://github.com/

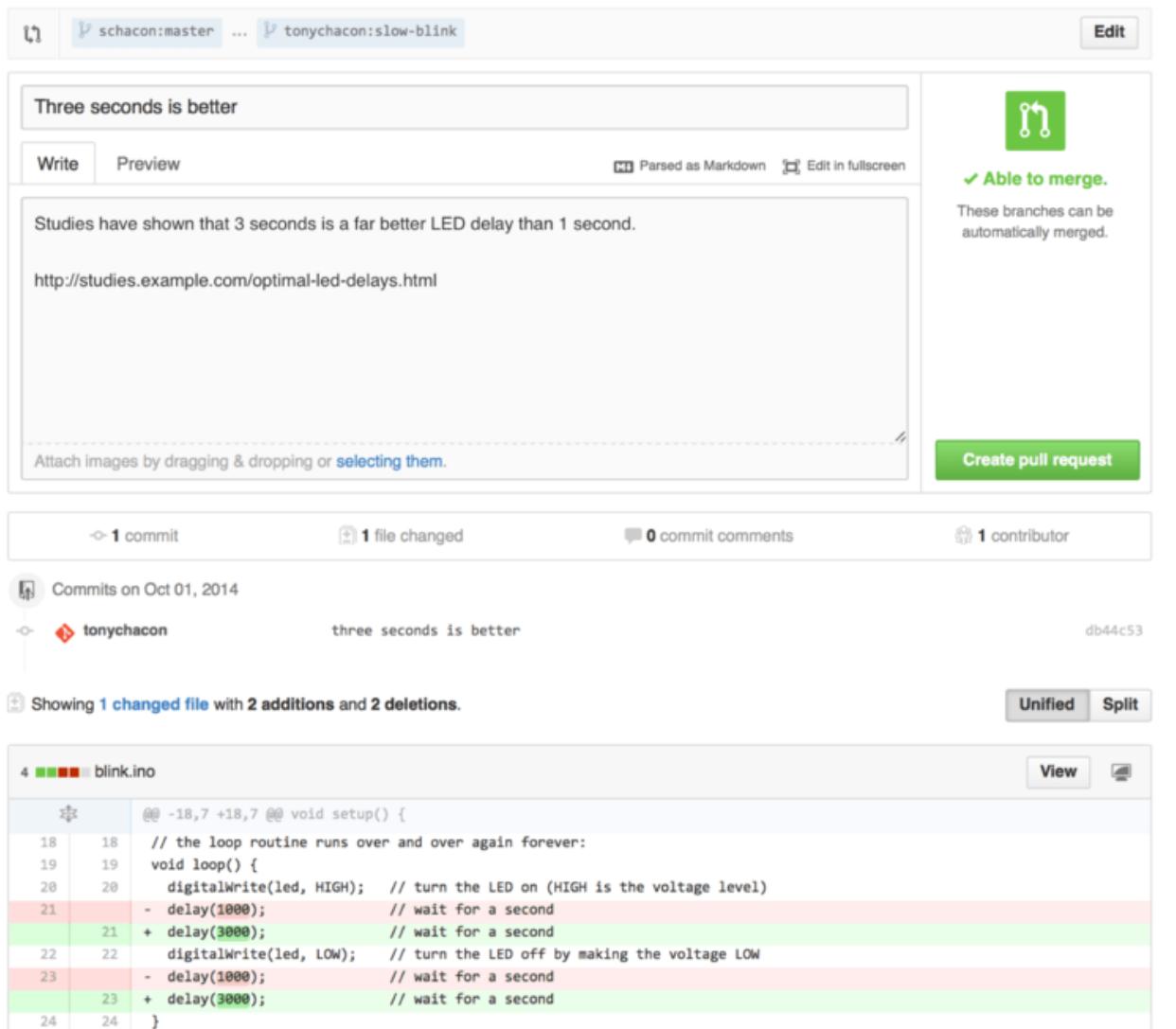
You can clone with [HTTPS](#), [SSH](#), or [Subversion](#).

Clone in Desktop Download ZIP

Figura 91. 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 buena descripción 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.



The screenshot shows a GitHub pull request page. At the top, it says "Three seconds is better". Below that are tabs for "Write" and "Preview". The "Preview" tab is selected, showing the content: "Studies have shown that 3 seconds is a far better LED delay than 1 second." followed by a link "http://studies.example.com/optimal-led-delays.html". To the right, there's a green icon with a gear and wrench, and the text "Able to merge. These branches can be automatically merged." A "Create pull request" button is at the bottom right of this section.

Below the preview, there are stats: "1 commit", "1 file changed", "0 commit comments", and "1 contributor". The commit details show "Commits on Oct 01, 2014" by "tonychacon" with the message "three seconds is better" and hash "db44c53".

At the bottom, it says "Showing 1 changed file with 2 additions and 2 deletions." with "Unified" and "Split" options. The diff view shows changes in "blink.ino":

```

diff --git a/blink.ino b/blink.ino
@@ -18,7 +18,7 @@ void setup() {
 18   18   // the loop routine runs over and over again forever:
 19   19   void loop() {
 20     20     digitalWrite(led, HIGH); // turn the LED on (HIGH is the voltage level)
 21     21     - delay(1000); // wait for a second
 22     22     + delay(3000); // wait for a second
 23     23     digitalWrite(led, LOW); // turn the LED off by making the voltage LOW
 24     24     - delay(1000); // wait for a second
 25     25     + delay(3000); // wait for a second
 26   }

```

Figura 92. 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.

NOTA

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 [Git en entornos distribuidos](#), 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”.

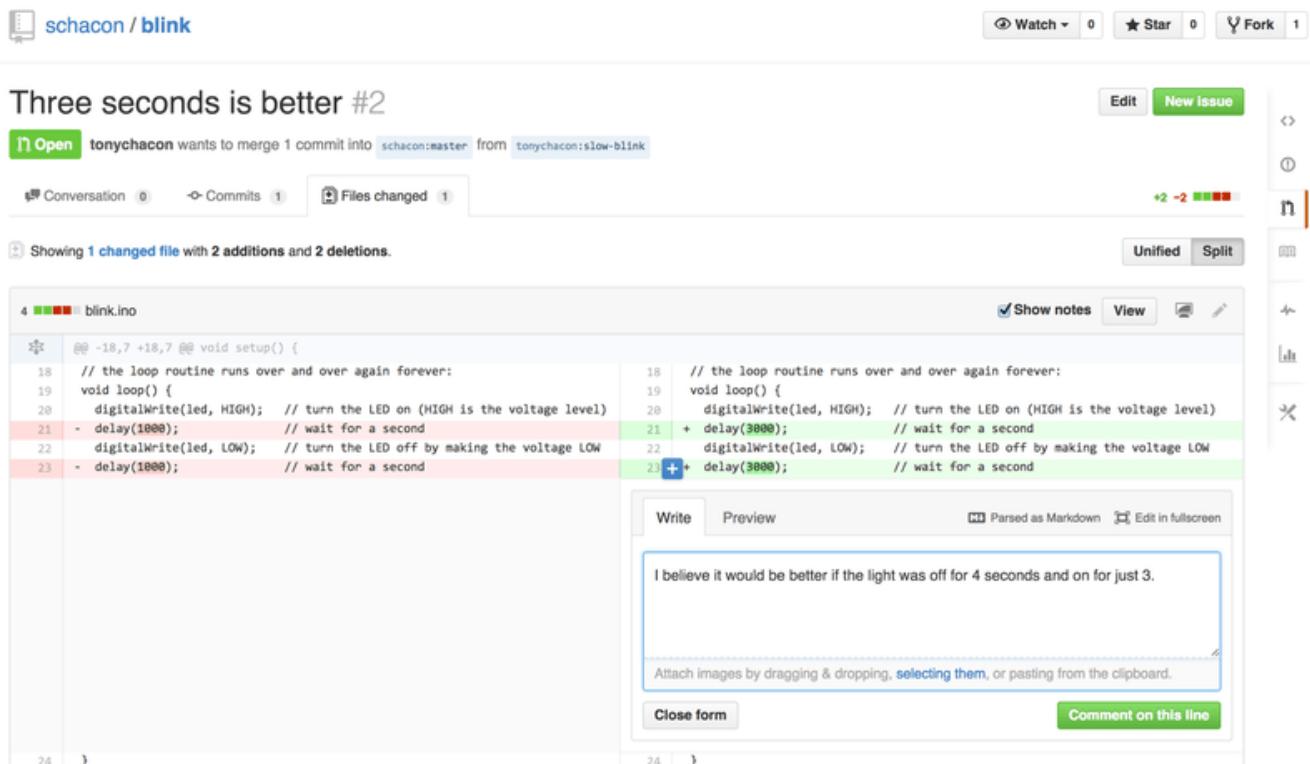


Figura 93. Comentando una línea concreta del diff

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:



Figura 94. Comentarios enviados en notificaciones de correo

Cualquiera puede añadir sus propios comentarios. En [Página de discusión del Pull Request](#) vemos un ejemplo de propietario de proyecto comentando tanto una línea del código como dejando un comentario general en la sección de discusión. Puedes comprobar que los comentarios del código se insertan igualmente en la conversación.

Three seconds is better #2

Edit New issue

Open tonychacon wants to merge 1 commit into schacon:master from tonychacon:slow-blink

Conversation 1 Commits 1 Files changed 1 +2 -2

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

schacon commented on the diff just now

blink.ino View full changes

```
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 Unsubscribe You're receiving notifications because you commented.

2 participants

Lock pull request

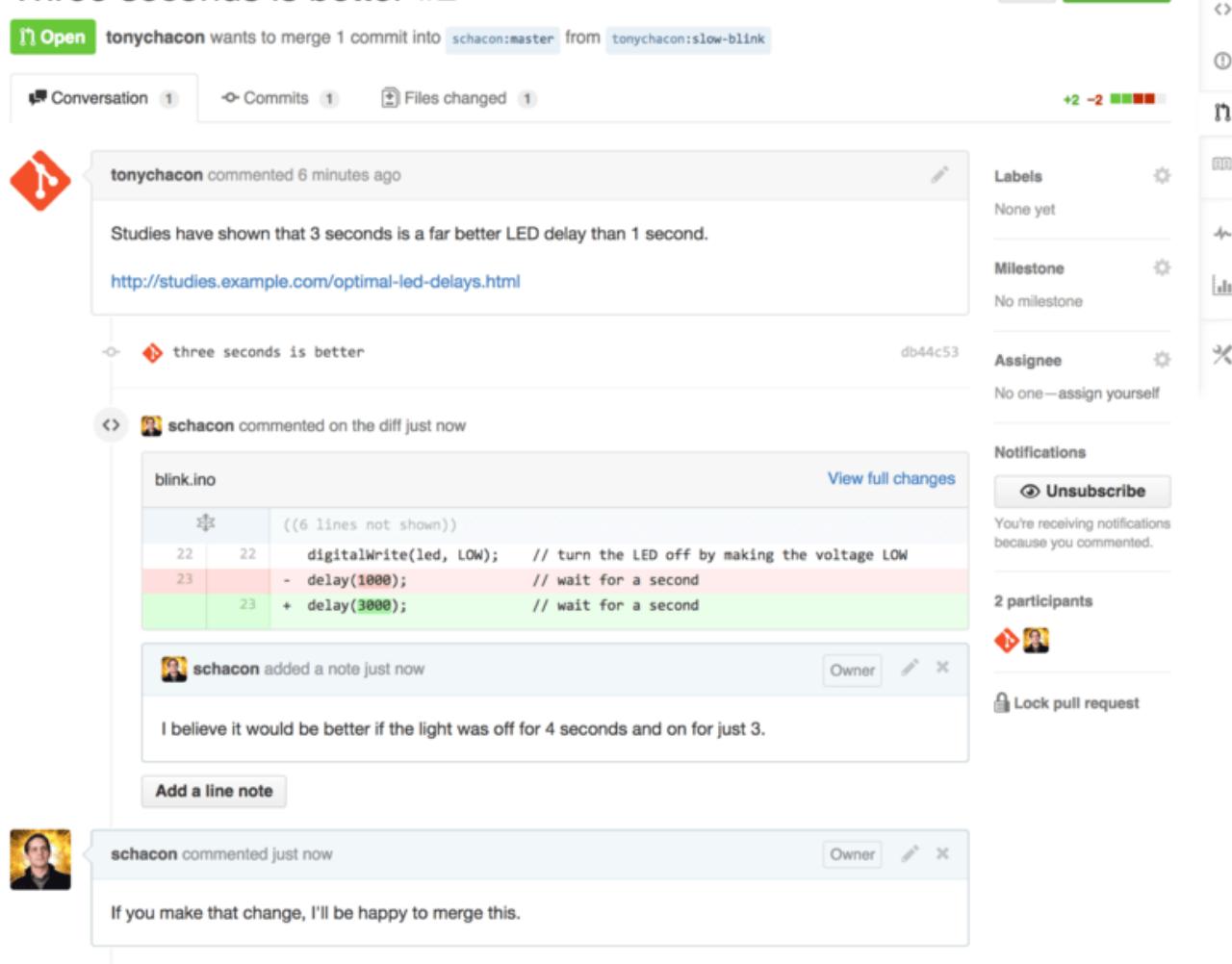


Figura 95. Página de discusión del Pull Request

El participante puede ver ahora qué tiene que hacer para que sea aceptado su cambio. Con suerte será 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 titled "Three seconds is better" by tonychacon. The pull request has 3 commits and 1 file changed. The conversation shows tonychacon commenting that studies show 3 seconds is better than 1 second, linking to a study page. schacon comments on an outdated diff. tonychacon adds some commits, mentioning longer off time and removing trailing whitespace. tonychacon then comments again, saying they changed it to 4 seconds and removed whitespace. A note at the bottom says the pull request can be automatically merged, with a "Merge pull request" button.

tonychacon commented 11 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

schacon commented on an outdated diff 5 minutes ago Show outdated diff

schacon commented 5 minutes ago

If you make that change, I'll be happy to merge this.

tonychacon added some commits 2 minutes ago

longer off time 0c1f66f
remove trailing whitespace ef4725c

tonychacon commented 10 seconds ago

I changed it to 4 seconds and also removed some trailing whitespace that I found. Anything else you would like me to do?

This pull request can be automatically merged. You can also merge branches on the command line.

Merge pull request

Figura 96. 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

NOTA

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 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 [Pull Request final](#), verás que el colaborador no reorganiza su commit y envía un nuevo Pull Request. En su 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.



Figura 97. Pull Request que no puede fusionarse limpiamente

Si ves algo parecido a [Pull Request que no puede fusionarse limpiamente](#), 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 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 por lejos más complicado de hacer y con mayores posibilidades de error.

Si quieres fusionar en la rama objetivo para hacer que tu Pull Request sea fusionable, deberías añadir el repositorio original como un nuevo remoto, bajártelo (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 re-chequea para ver si es posible un fusionado automático o no.

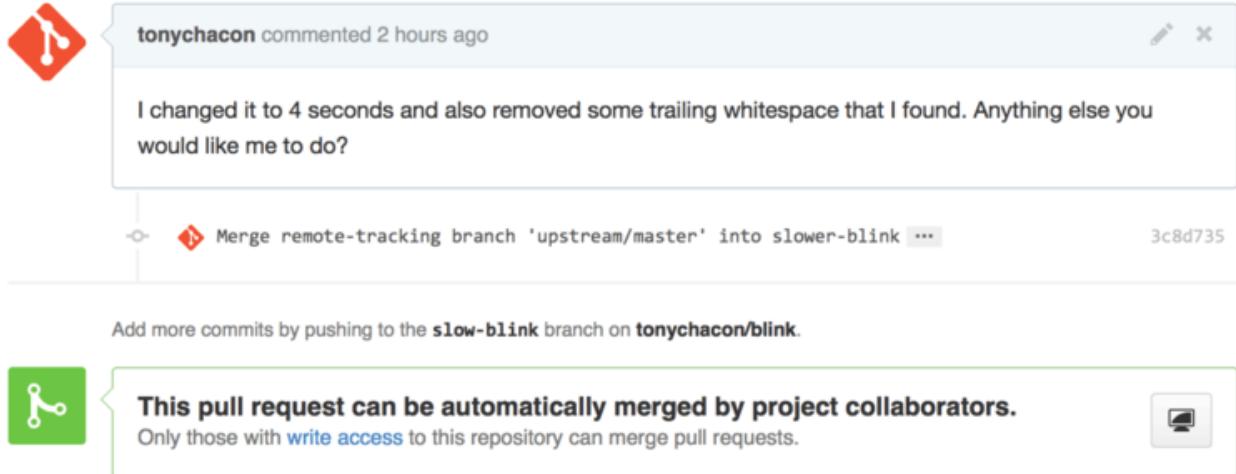


Figura 98. Ahora el Pull Request ya fusiona bien

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](#)) 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 último.

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 hacer 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 [Referencias cruzadas en un Pull Request..](#)

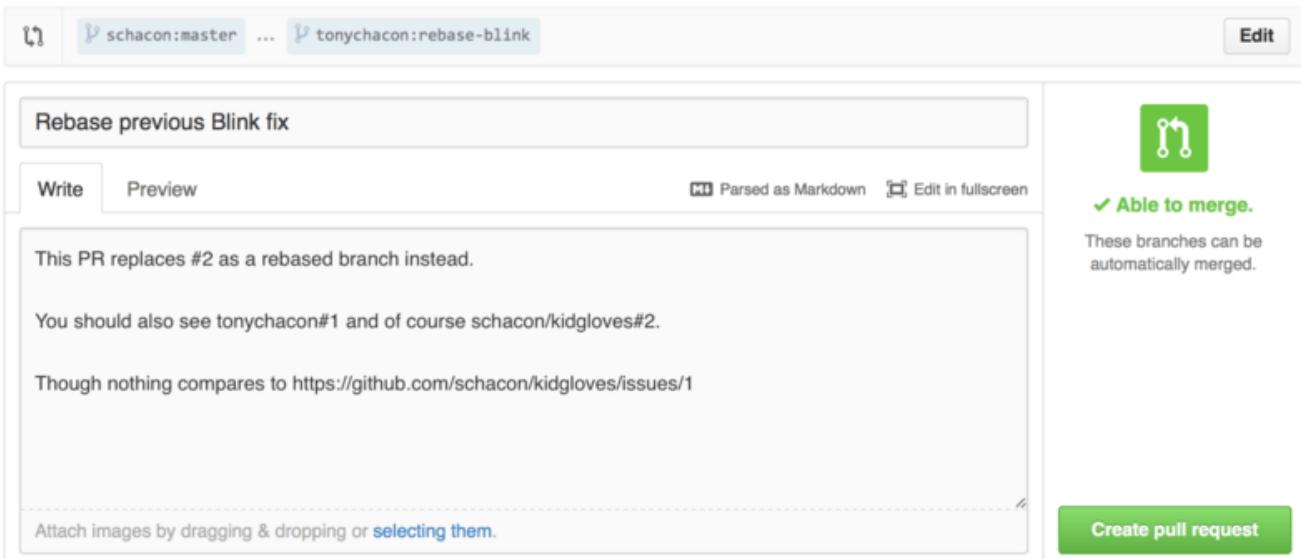


Figura 99. Referencias cruzadas en un Pull Request.

Cuando enviamos este Pull Pequest, veremos todo como en [Cómo se ven las referencias cruzadas en el Pull Request..](#)

Rebase previous Blink fix #4

The screenshot shows a GitHub Pull Request page for a merge from `tonychacon:rebase-blink` into `schacon:master`. The title is 'Rebase previous Blink fix #4'. A comment from `tonychacon` is visible, along with a commit history showing two commits added by `tonychacon` 4 hours ago, with SHA codes `afe904a` and `a5a7751`.

Figura 100. 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 cualquiera 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 [Cómo se ven las referencias cruzadas en el Pull Request..](#)

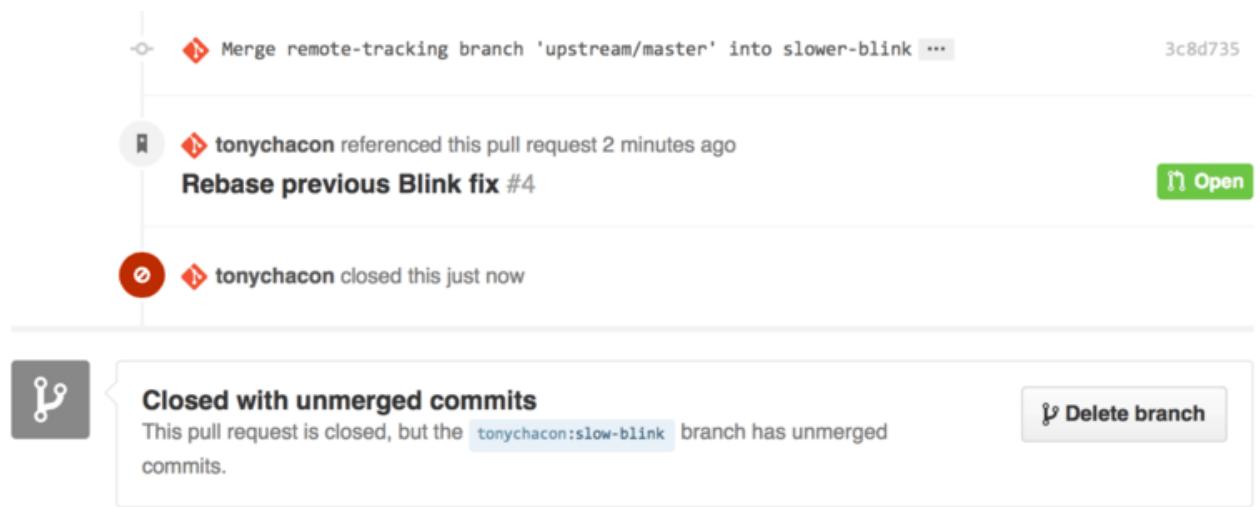


Figura 101. Cómo se ven las referencias cruzadas en el Pull Request.

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

El enlazado a otras incidencias es sólo 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 como “formato Markdown de GitHub”. El formato Markdown es como escribir en texto plano pero que luego se convierte en texto con formato.

Mira en [Ejemplo de texto en Markdown y cómo queda después](#) un ejemplo de cómo los comentarios o el texto puede escribirse y luego formatearse con Markdown.

The left side shows a GitHub Markdown editor window with a preview pane. It contains a text block about a problem with the blink code, mentioning the number 13 and its issues. The right side shows a GitHub comment from tonychacon where the text has been converted into a formatted list and a bolded statement.

Figura 102. Ejemplo de texto en Markdown y cómo queda después.

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 a realizar 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 [Cómo se ven las listas de tareas de Markdown](#).



Figura 103. Cómo se ven las listas de tareas de Markdown.

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 de 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 [Resumen de lista de tareas en la lista de PR..](#)

Figura 104. 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.

Fragmentos de 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);
}
```

```

Siañades 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 [Cómo se ve el fragmento de código..](#)

Figura 105. Cómo se ve el fragmento de código.

Citas

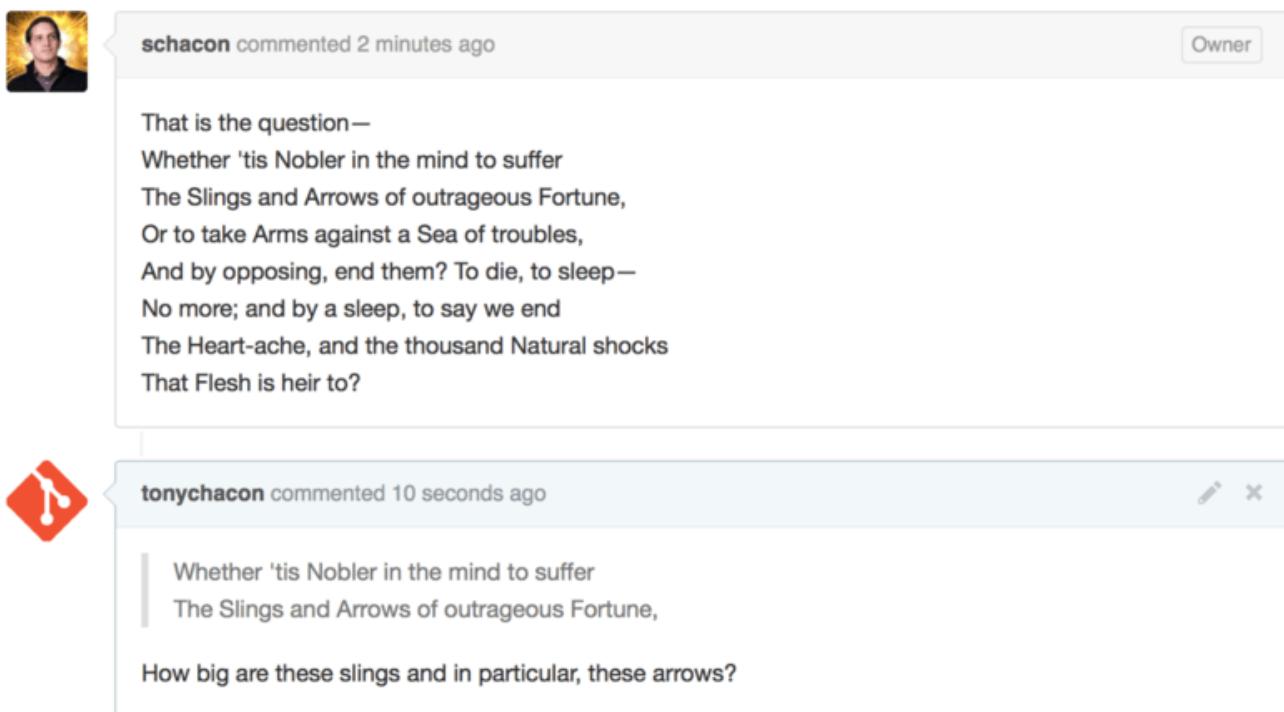
Si estás respondiendo a un comentario grande, pero solo a una pequeña parte, puedes seleccionar la parte que te interesa y citarlo, para lo que precedes cada línea citada del símbolo >. Esto es tan útil que hay un atajo de teclado para hacerlo: si seleccionas el

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 introducido, el comentario se vería como en [Rendered quoting example..](#)



The screenshot shows two GitHub comment threads. The top thread is by user **schacon**, who commented 2 minutes ago with a quote from Shakespeare's Hamlet. The quote is:

That is the question—
Whether 'tis Nobler in the mind to suffer
The Slings and Arrows of outrageous Fortune,
Or to take Arms against a Sea of troubles,
And by opposing, end them? To die, to sleep—
No more; and by a sleep, to say we end
The Heart-ache, and the thousand Natural shocks
That Flesh is heir to?

The bottom thread is by user **tonychacon**, who commented 10 seconds ago with a quote from the same source:

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?

Both comments have a small red diamond icon with a white gear and wrench symbol to the left of the author's name. There are edit and delete icons at the top right of each comment box.

Figura 106. Rendered quoting example.

Emojis (emoticonos)

Finalmente, también puedes usar emojis (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.

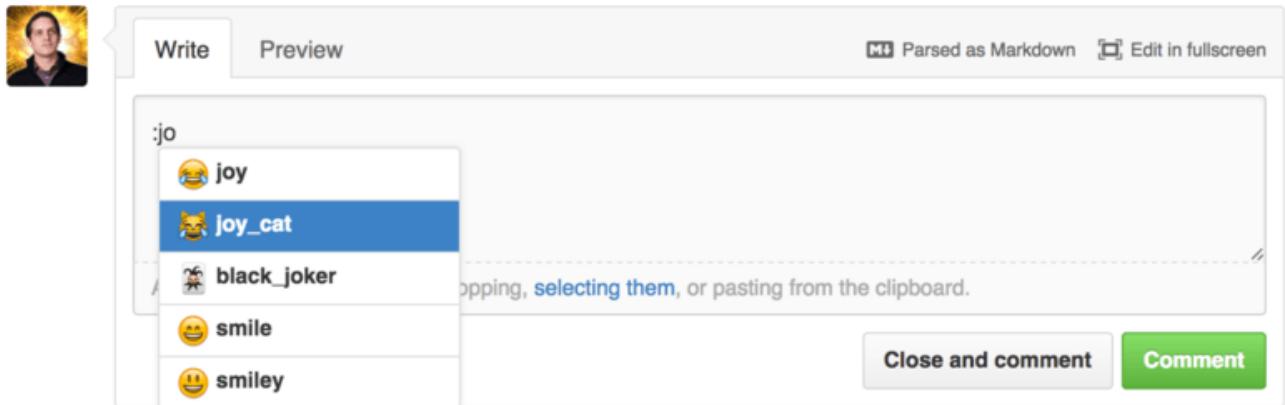


Figura 107. Emoji auto-completando emoji.

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 [Comentando con muchos emoji..](#)

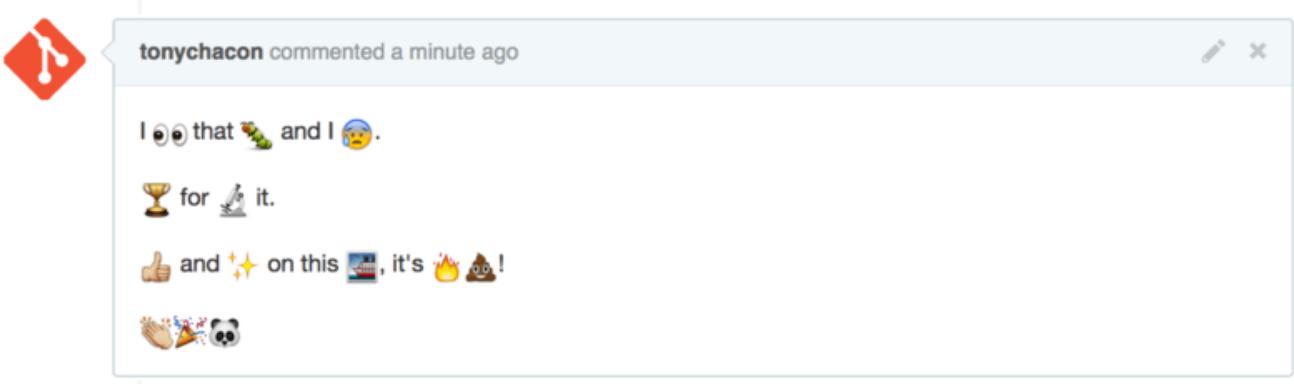


Figura 108. Comentando con muchos emoji.

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.

NOTA

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.

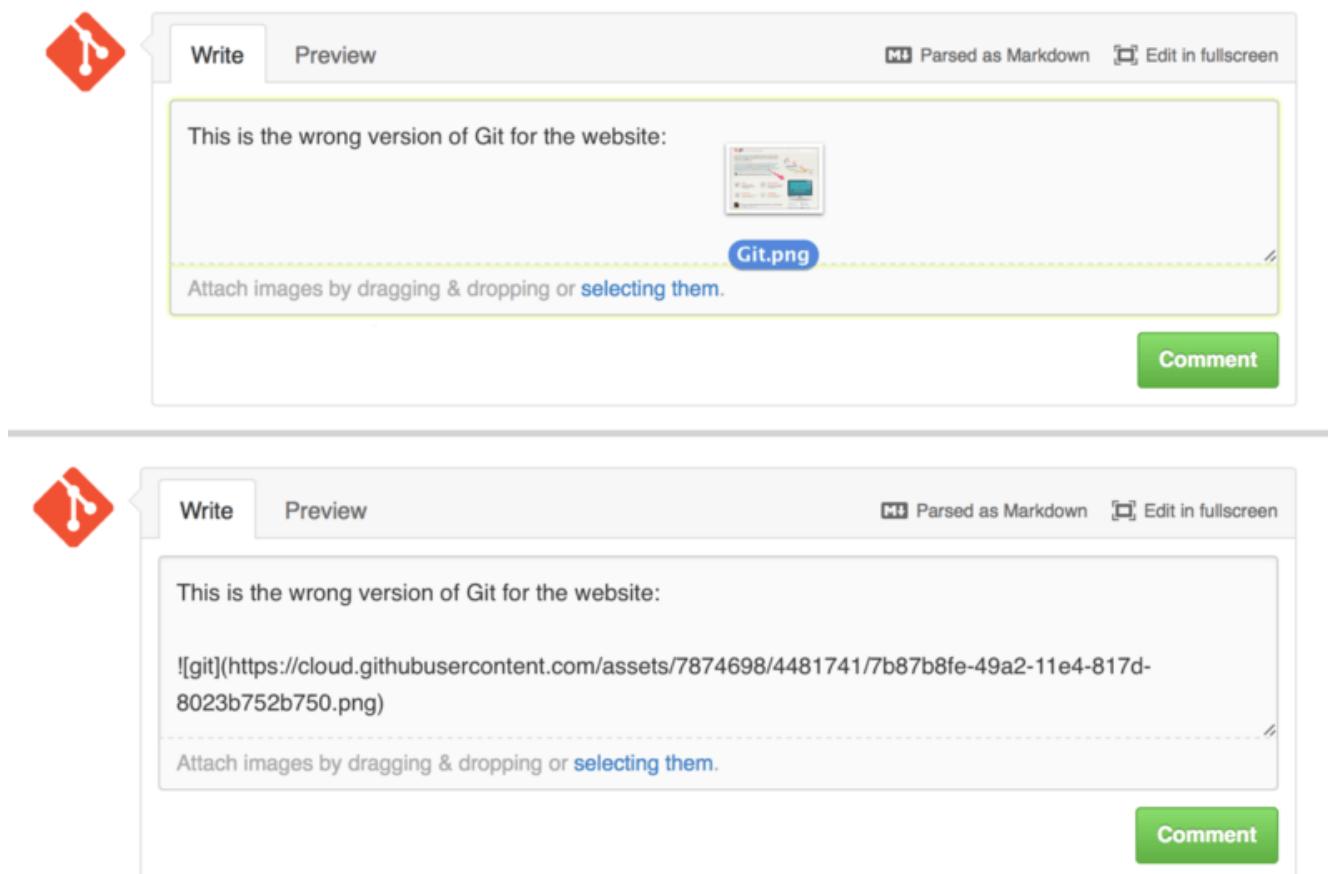


Figura 109. Arrastrar y soltar imágenes para subirlas.

Si vuelves a [Referencias cruzadas en un Pull Request.](#), 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 [Desplegable “New repository”..](#)

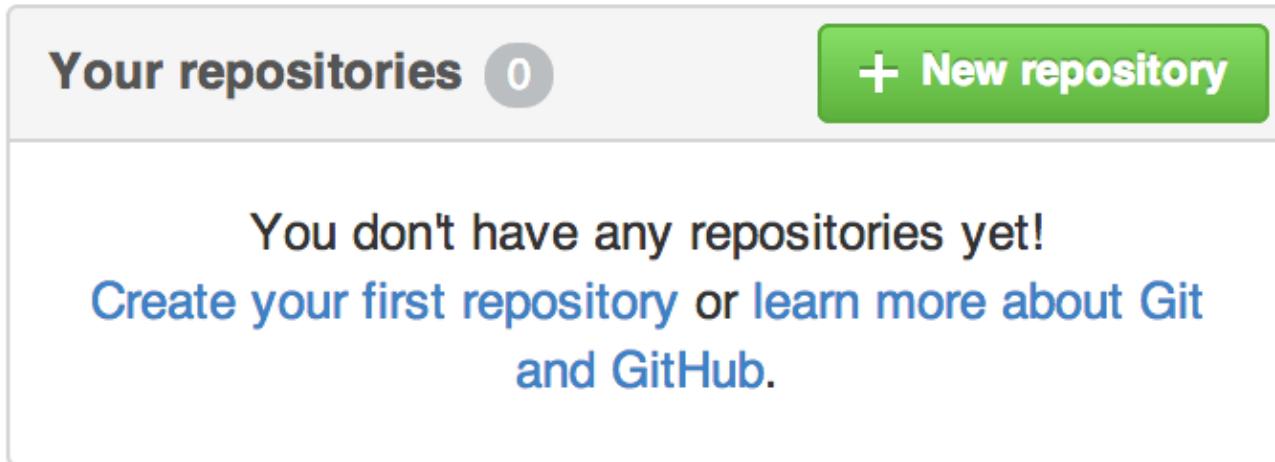


Figura 110. La zona “Your repositories”.

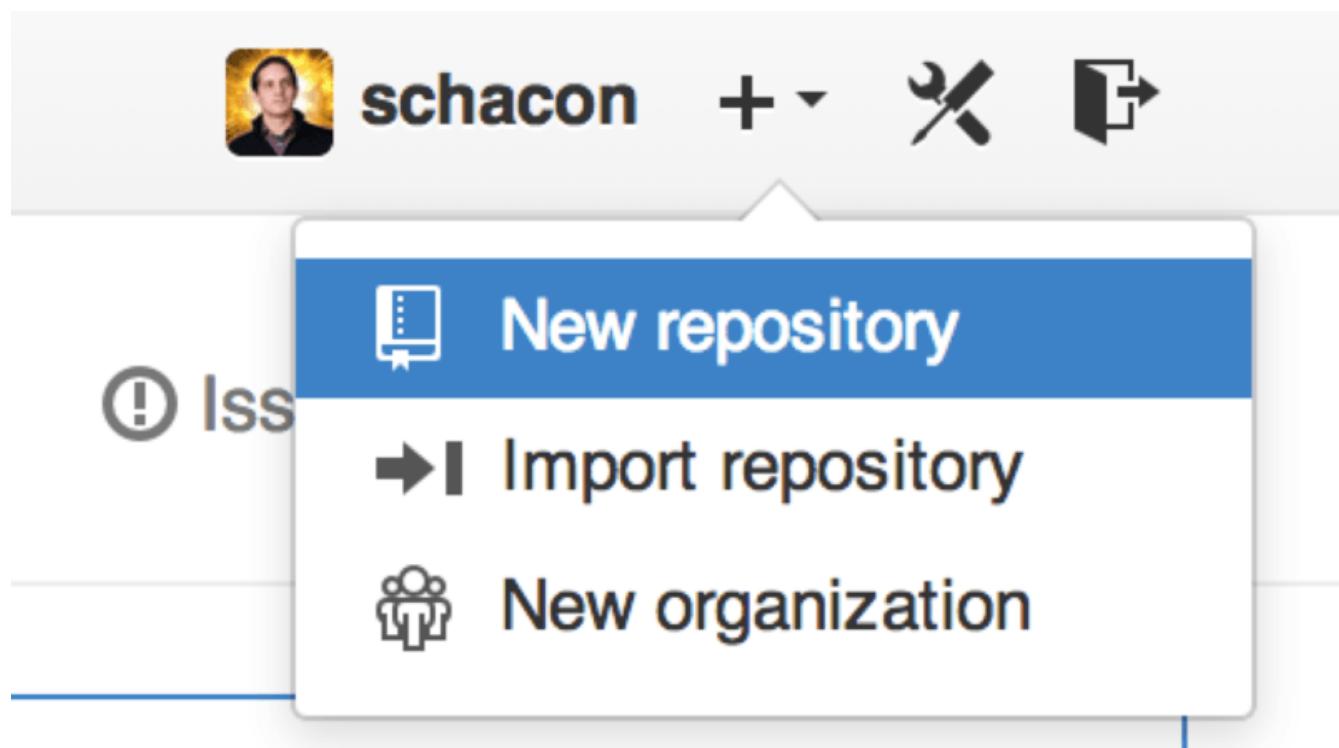


Figura 111. Desplegable “New repository”.

Esto te llevará al formulario para crear un nuevo repositorio:

Owner Repository name

PUBLIC  ben / iOSApp 

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

Description (optional)

iOS project for our mobile group

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

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

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

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

Create repository

Figura 112. Formulario para crear repositorio.

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 [Fundamentos de Git](#).

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 HTTPS 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.

NOTA

Suele ser preferible compartir la URL de tipo HTTPS 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 HTTPS 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.

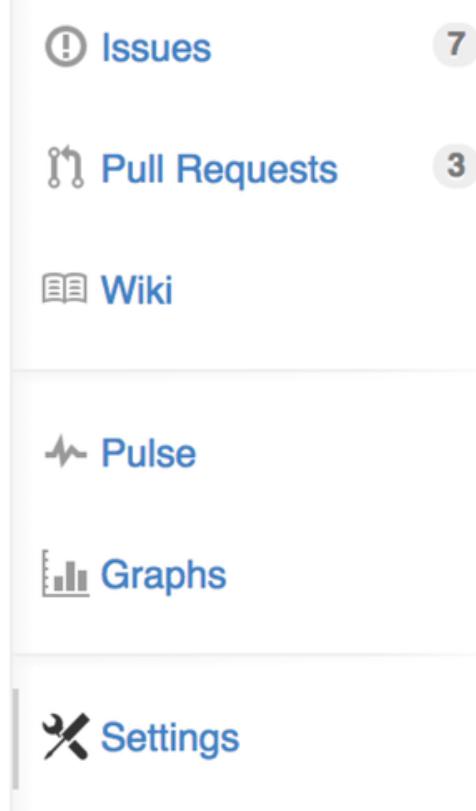


Figura 113. Enlace a ajustes del repositorio.

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. Recuerda que si el proyecto está en un repositorio privado gratuito, solo podrás dar accesos a tres colaboradores. Si necesitas quitar un acceso, pulsa en la “X” del lado derecho del usuario.

| Options | Collaborators | Full access to the repository |
|----------------------|--|-------------------------------|
| Collaborators | Ben Straub ben | x |
| Webhooks & Services | Jeff King peff | x |
| Deploy keys | Louise Corrigan LouiseCorrigan | x |
| | Type a username | Add collaborator |

Figura 114. 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 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 de escritura 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 [Notificación por correo de nuevo Pull Request..](#)

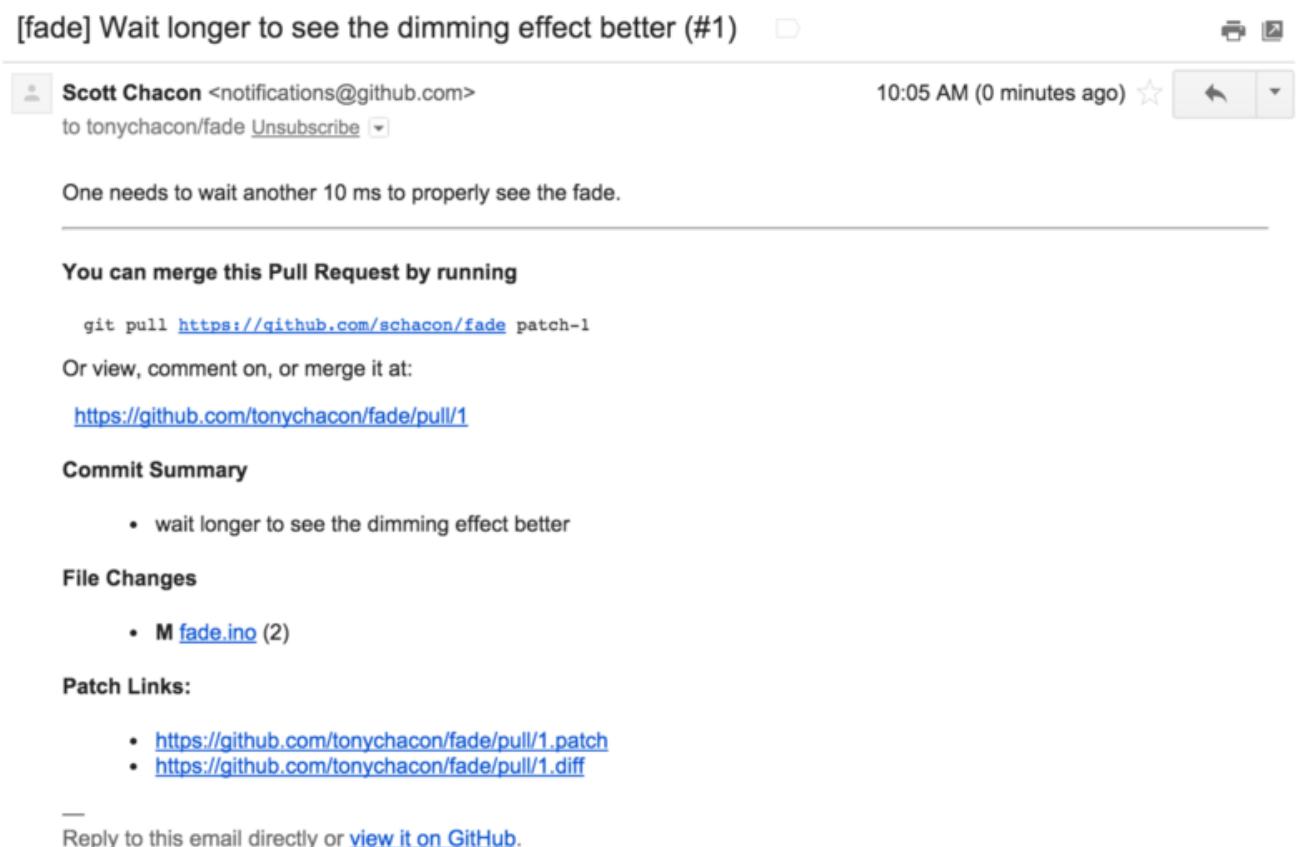


Figura 115. Notificación por correo de nuevo Pull Request.

Hay algunas cosas a destacar en este correo. En primer lugar, te dará un pequeño *diffstat* (es decir, una lista de archivos cambiados y en qué medida). Además, 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 lo indica, proporcionan “diff unificados” y formatos de parche del Pull Request. Técnicamente, podrías fusionar con algo como:

```
$ curl https://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.

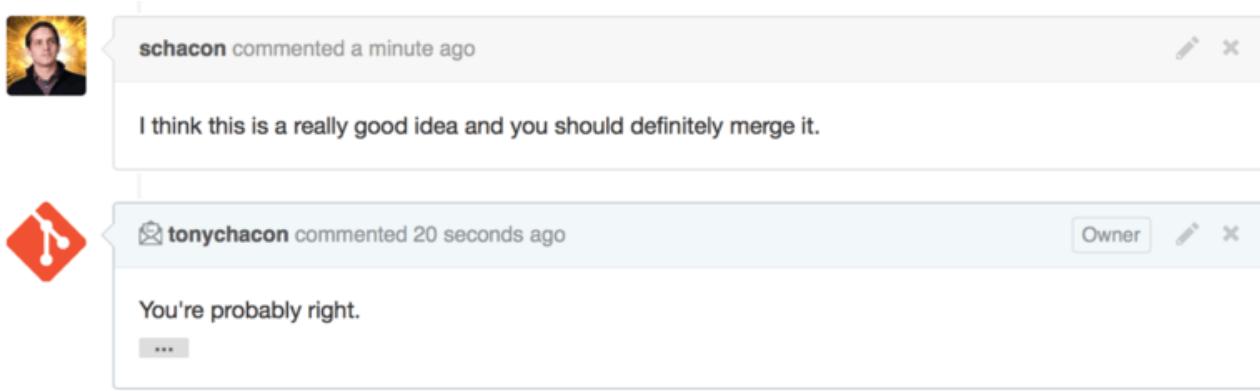


Figura 116. Las respuestas a correos se incluyen en el hilo de discusión.

Una vez que el código está como quieras y deseas 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 incluso 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 [Botón Merge e instrucciones para fusionar manualmente un Pull Request](#), GitHub te da toda esta información si pulsas en el enlace de ayuda.

This pull request can be automatically merged.
You can also merge branches on the [command line](#).

Merging via command line
If you do not want to use the merge button or an automatic merge cannot be performed, you can perform a manual merge on the command line.

HTTP Git Patch <https://github.com/schacon/fade.git>

Step 1: From your project repository, check out a new branch and test the changes.

```
git checkout -b schacon-patch-1 master  
git pull https://github.com/schacon/fade.git patch-1
```

Step 2: Merge the changes and update on GitHub.

```
git checkout master  
git merge --no-ff schacon-patch-1  
git push origin master
```

Figura 117. 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 [Las especificaciones para hacer referencia a... \(refspec\)](#), 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 [Fontanería y porcelana](#)) 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
3c8d735ee16296c242be7a9742ebfbc2665adec1    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 archivo `.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 refspec:

```
[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 archivo 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.

The screenshot shows two GitHub pull request comparison pages. The top page compares 'schacon:master' and 'tonychacon:patch-2'. It displays 2 commits, 1 file changed, 0 commit comments, and 2 contributors. The bottom page shows a dropdown menu for 'base: patch-1' with 'master' selected. It also shows 1 commit, 0 commit comments, and 1 contributor. Both pages include a 'Create pull request' button and a 'Discuss' section.

Figura 118. 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 anteponiendo el carácter @, intentará auto-completar nombres de usuario de personas que sean colaboradores o responsables en el proyecto.

The screenshot shows the GitHub comment input interface. The 'Write' tab is active, and the preview area shows a list of users starting with '@'. A tooltip says 'selecting them, or pasting from the clipboard.' There are buttons for 'Close and comment' and 'Comment' at the bottom.

Figura 119. 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 “suscrito” 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.

Notifications

 **Unsubscribe**

You're receiving notifications because you commented.

Figura 120. Quitar suscripción de un pull request o incidencia.

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.

Figura 121. 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 [Centro de notificaciones..](#)

Figura 122. Centro de notificaciones.

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 su 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 icono 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 características 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 activas, recibirás los correos de cada notificación. Vimos ya algún ejemplo en [Comentarios enviados en notificaciones de correo](#) y [Notificación por correo de nuevo Pull Request](#). 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 adecuadas.

Por ejemplo, si miramos las cabeceras de los correos enviados a Tony en el correo visto en [Notificación por correo de nuevo Pull Request](#), 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** permiten 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.

Archivos especiales

Hay dos archivos especiales que GitHub detecta y maneja si están presentes en el repositorio.

README

En primer lugar tenemos el archivo **README**, que puede estar en varios formatos. Puede estar con el nombre **README**, **README.md**, **README.asciidoc** y alguno más. Cuando GitHub detecta su presencia en el proyecto, lo muestra en la página principal, con el *renderizado* que corresponda a su formato.

En muchos casos este archivo 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 archivo, puedes incluir imágenes o enlaces en él para facilitar su comprensión.

CONTRIBUTING

El otro archivo que GitHub reconoce es **CONTRIBUTING**. Si tienes un archivo con ese nombre y cualquier extensión, GitHub mostrará algo como [Apertura de un Pull Request cuando existe el archivo CONTRIBUTING](#) cuando se intente abrir un Pull Request.

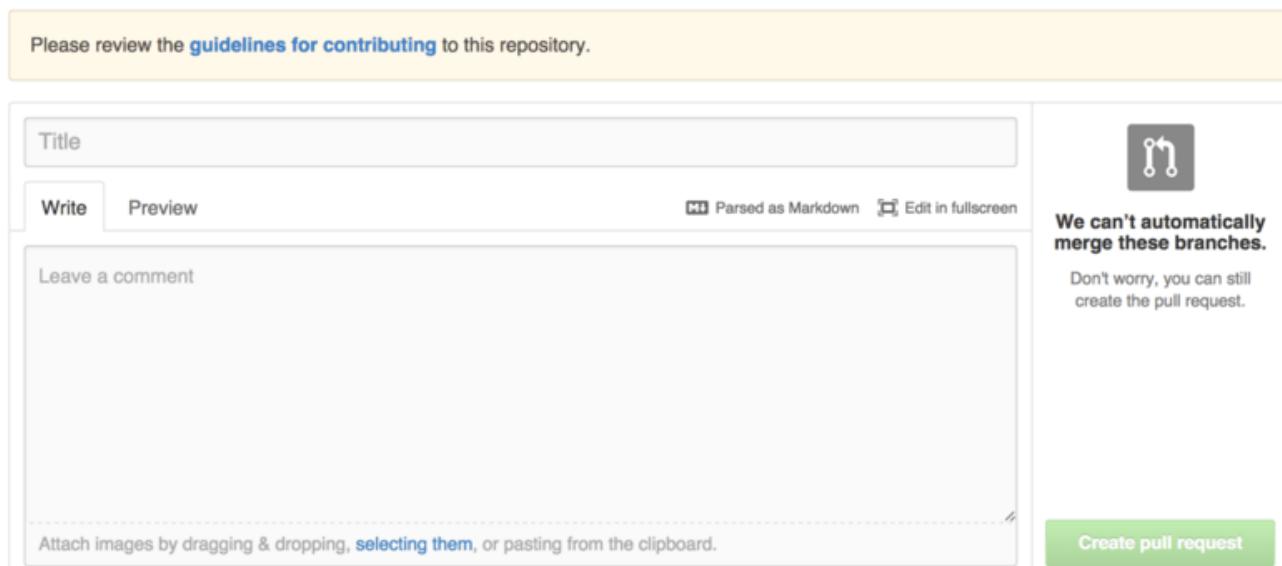


Figura 123. Apertura de un Pull Request cuando existe el archivo CONTRIBUTING.

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”.

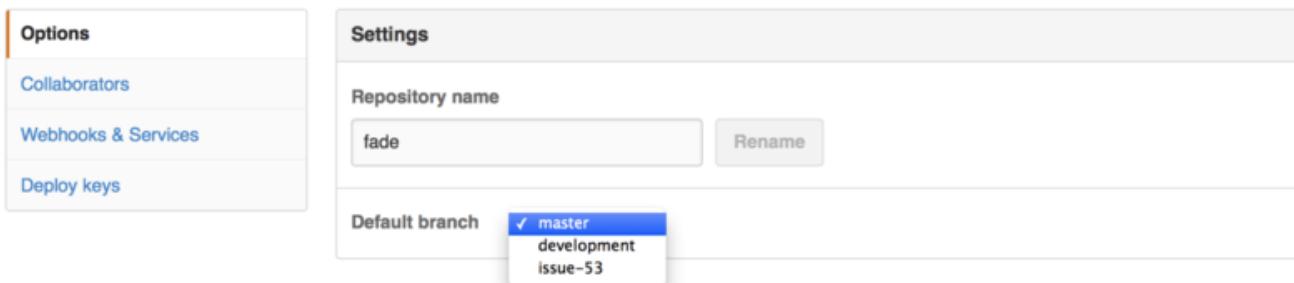


Figura 124. Cambio de la rama predeterminada del proyecto.

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 quieres transferir la propiedad de un proyecto a otro usuario u organización en GitHub, hay una opción para ello al final de “Options” llamada “Transfer ownership”.

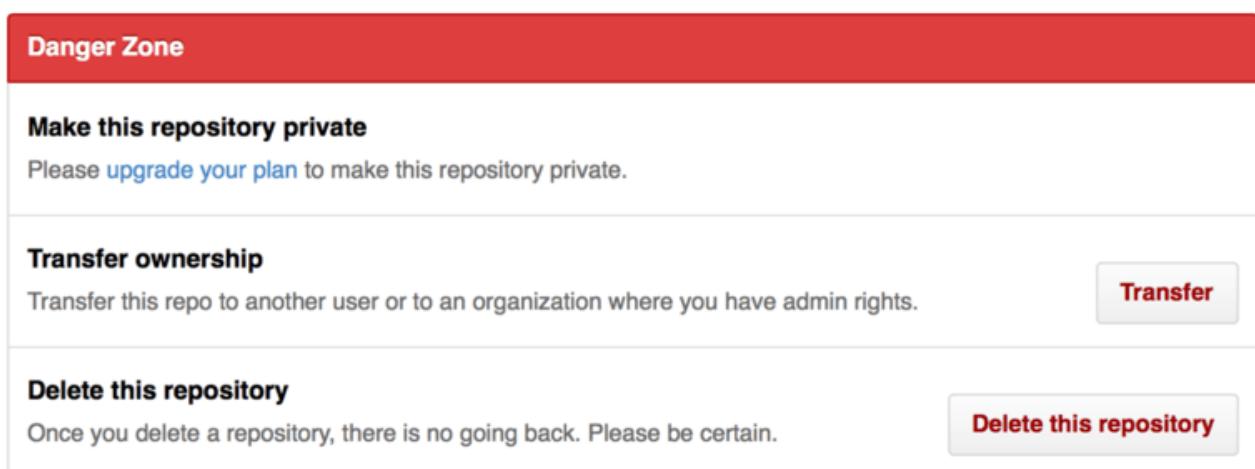


Figura 125. Transferir propiedad de un proyecto.

Esto es útil si vas a abandonar el proyecto y quieres 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”.

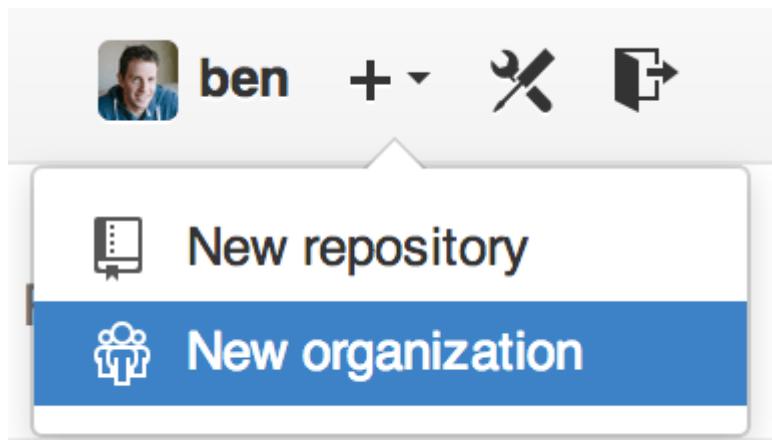


Figura 126. El menú “New organization”.

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. A diferencia 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 icono](#), 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 gear icon for settings. Below that is a search bar with 'Filters' and 'Find a repository...', and a green button '+ New repository'. The main area displays three repositories: '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 panels: 'People' (listing three members: dragonchacon, schacon, and tonychacon) and 'Teams' (listing three teams: Owners, Frontend Developers, and Ops).

| Team | Members | Repositories |
|---------------------|-----------|----------------|
| Owners | 1 member | 3 repositories |
| Frontend Developers | 2 members | 2 repositories |
| Ops | 3 members | 1 repository |

Figura 127. Página de la organización.

Para gestionar tus equipos, puedes pulsar en la barra “Teams” del lado derecho en la página [Página de la organización](#). Esto te llevará a una página en 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 de solo lectura, de escritura o administrativo al repositorio. Puedes cambiar el nivel pulsando en el botón “Settings” en [Página de equipos](#).

The screenshot shows the GitHub interface for a team named 'Frontend Developers'. On the left, there's a sidebar with team statistics: 2 MEMBERS and 2 REPOSITORIES. Below that are 'Leave' and 'Settings' buttons. The main area has tabs for 'Members' (which is selected) and 'Repositories'. Under 'Members', there are two entries: 'tonychacon' (Tony Chacon) and 'schacon' (Scott Chacon). Each entry includes a small profile picture, the member's name, their GitHub handle, and a 'Remove' button. A button at the top right says 'Invite or add users to team'.

This team grants **Admin** access: members can read from, push to, and add collaborators to the team's repositories.

Figura 128. Página de equipos.

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 quieres involucrar a un equipo en algo al no tener claro a quién en 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 usarlos 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.



| Recent events | | Filters ▾ | | |
|---------------|---|-----------|--------|----------------|
| | | Search... | | |
| dragonchacon | added themselves to the chaoncorp/ops team | | member | 32 minutes ago |
| schacon | added themselves to the chaoncorp/ops team | | | |
| tonychacon | invited dragonchacon to the chaoncorp organization | | member | 33 minutes ago |
| tonychacon | invited schacon to the chaoncorp organization | | | |
| tonychacon | gave chaoncorp/ops access to chaoncorp/backend | | member | 16 hours ago |
| tonychacon | gave chaoncorp/frontend-developers access to chaoncorp/backend | | | |
| tonychacon | gave chaoncorp/frontend-developers access to chaoncorp/frontend | | | 16 hours ago |
| tonychacon | created the repository chaoncorp/deployscripts | | | 16 hours ago |
| tonychacon | created the repository chaoncorp/backend | | | 16 hours ago |

Figura 129. 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 *hackable* 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 [Sección Services and Hooks..](#)

The screenshot shows the 'Services and Hooks' section of a GitHub repository's settings. On the left, a sidebar lists 'Options', 'Collaborators', 'Webhooks & Services' (which is selected and highlighted in orange), and 'Deploy keys'. The main area has two tabs: 'Webhooks' and 'Services'. The 'Webhooks' tab contains a brief description of what webhooks are and how they work, with a 'Add webhook' button. The 'Services' tab contains a brief description of what services are and how they perform actions, with a 'Add service' button. A modal window is open over the 'Services' tab, titled 'Available Services'. It shows a search input with 'email' typed in and a blue button labeled 'Email' at the bottom.

Figura 130. 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 [Configuración de servicio de correo..](#)

Options

Collaborators

Webhooks & Services

Deploy keys

Services / Add Email

Install Notes

- address whitespace separated email addresses (at most two)
- secret fills out the Approved header to automatically approve the message in a read-only or moderated mailing list.
- `send_from_author` uses the commit author email address in the From address of the email.

Address

Secret

Send from author

Active
We will run this service when an event is triggered.

Add service

Figura 131. 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 sólo 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 quieres 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 [Sección Services and Hooks..](#) Esto mostrará una página como [Configuración de enganches web..](#)

The screenshot shows the GitHub settings interface for managing webhooks. On the left, a sidebar lists 'Options', 'Collaborators', 'Webhooks & Services' (which is selected and highlighted in orange), and 'Deploy keys'. The main content area is titled 'Webhooks / Add webhook'. It contains instructions about sending POST requests to the specified URL with event details. A 'Payload URL' field is populated with 'https://example.com/postreceive'. The 'Content type' dropdown is set to 'application/json'. A 'Secret' field is present but empty. Below these, a section asks 'Which events would you like to trigger this webhook?' with three options: 'Just the push event.' (selected), 'Send me everything.', and 'Let me select individual events.'. A checked 'Active' checkbox indicates the hook will deliver event details when triggered. At the bottom is a green 'Add webhook' button.

Figura 132. 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**, que se dispara 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 archivo 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 archivos 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 pueden ver las últimas veces que GitHub ha intentado ejecutar el enganche. Para cada uno, puedes mirar qué información se ha enviado y si fué recibido correctamente, junto con las cabeceras correspondientes de la petición y de la respuesta. Esto facilita mucho las pruebas de tus enganches.

Recent Deliveries

| | | | |
|--------------------------------------|--------------------------------------|---------------------|-----|
| ⚠ | 4aeae280-4e38-11e4-9bac-c130e992644b | 2014-10-07 17:40:41 | ... |
| ✓ | aff20880-4e37-11e4-9089-35319435e08b | 2014-10-07 17:36:21 | ... |
| ✓ | 90f37680-4e37-11e4-9508-227d13b2ccfc | 2014-10-07 17:35:29 | ... |

Request Response 200

🕒 Completed in 0.61 seconds. ↻ Redeliver

Headers

```
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
```

Payload

```
{
  "ref": "refs/heads/remove-whitespace",
  "before": "99d4fe5bfffaf827f8a9e7cde00ccb0ab06a35e48",
  "after": "9370a6c3349331bac7e4c3c78c10bc8460c1e3e8",
  "created": false,
  "deleted": false,
  "forced": false,
  "base_ref": null,
  "compare": "https://github.com/tonychacon/fade/compare/99d4fe5bfffaf...9370a6c33493",
  "commits": [
    {
      "id": "9370a6c3349331bac7e4c3c78c10bc8460c1e3e8",
      "distinct": true,
      "message": "remove whitespace",
      "timestamp": "2014-10-07T17:35:22+02:00",
      "url": "https://github.com/tonychacon/fade/commit/9370a6c3349331bac7e4c3c78c10bc8460c1e3e8"
    }
  ]
}
```

Figura 133. Depuración de un web hook.

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 de 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 archivo Markdown cualquiera 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.

The screenshot shows the GitHub user profile page for 'tonychacon'. On the left, there's a sidebar with links: Profile, Account settings, Emails, Notification center, Billing, SSH keys, Security, Applications (which is selected), Repositories, and Organizations. Below the sidebar, there's a section for 'chaconcorp' with a GitHub icon. The main content area has three sections: 'Developer applications', 'Personal access tokens', and 'Authorized applications'. The 'Personal access tokens' section contains a 'Generate new token' button and a note about generating an API token for scripts or testing. It also includes a detailed explanation of what personal access tokens are. The 'Authorized applications' section notes that there are no authorized apps. The 'GitHub applications' section lists 'GitHub Team' with a 'Last used on Oct 6, 2014' timestamp and a 'Revoke' button.

Figura 134. 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 ventaja 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 [Comentario enviado desde la API de GitHub..](#)

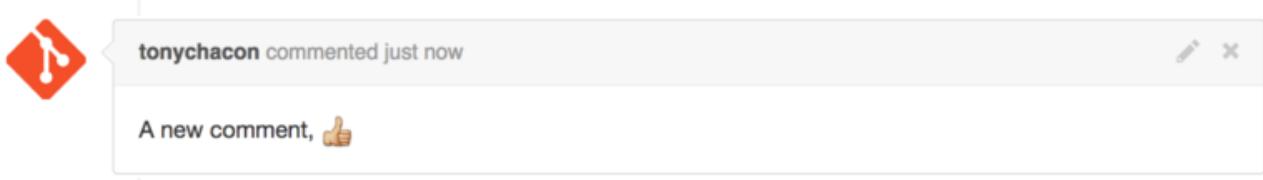


Figura 135. Comentario enviado desde la API de GitHub.

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 él, 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 'httparty'
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>](https://api.github.com/repos/<user>/<repo>/statuses/<commit_sha>) con el resultado.

En este caso, puedes enviar un estado (*success*, *failure*, *error*), una descripción de qué ocurrió, una 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 [Estado del commit mediante API](#).

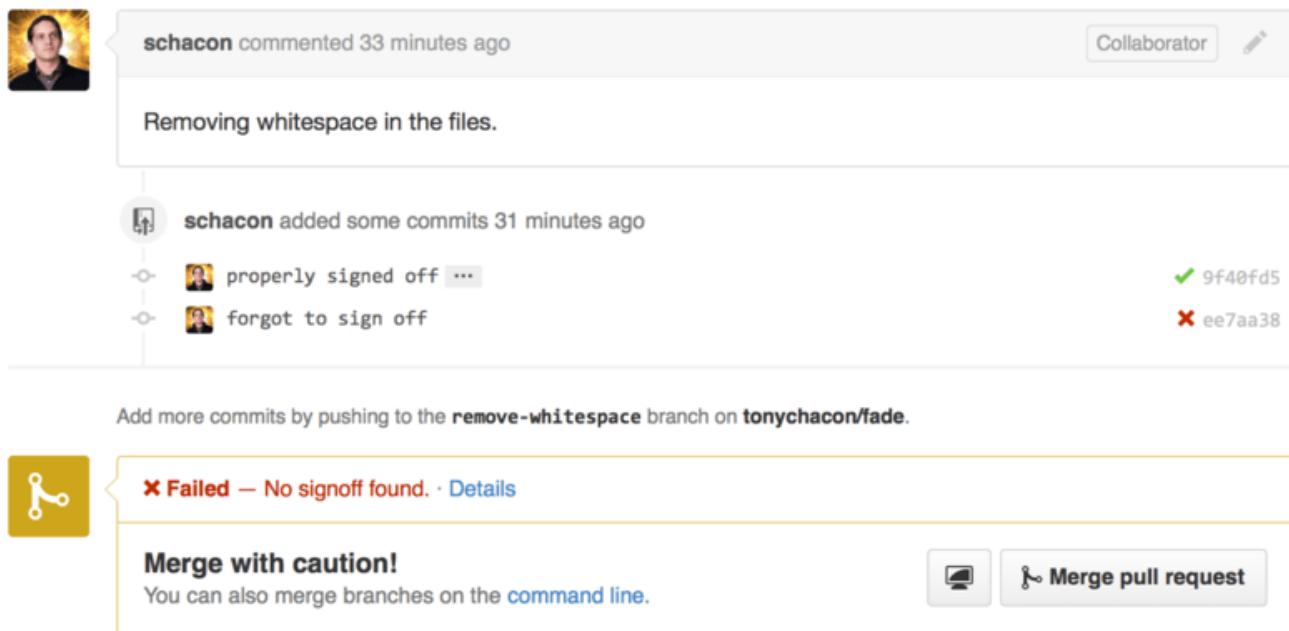


Figura 136. Estado del commit mediante API.

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 <https://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 integrarlo 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.

Herramientas de Git

Hasta ahora, ya has aprendido la mayoría de los comandos diarios y el flujo de trabajo que necesitas para manejar y mantener un repositorio de Git para tu control del código fuente. Has conseguido cumplir con las tareas básicas de seguimiento y has agregado archivos, además has aprovechado el poder del área de staging y has conocido el tema de branching y merging.

Ahora vas a explorar unas cuantas cosas bastante poderosas que Git puede realizar y que no necesariamente vas a usar en tu día a día, pero que puedes necesitar en algún momento.

Revisión por selección

Git te permite especificar ciertos *commits* o un rango de éstos de muchas maneras. No son necesariamente obvias, pero es útil conocerlas.

Revisiones individuales

Obviamente se puede referir a un “commit” por el hash SHA-1 que se le asigna, pero también existen formas más amigables de referirse a los *commits*. Esta sección delinea varias maneras en las que se puede referir a un “commit” individual.

SHA-1 corto

Git es lo suficientemente inteligente como para descifrar el “commit” al que te refieres si le entregas los primeros caracteres, siempre y cuando la parte de SHA-1 sea de al menos 4 caracteres y no sea ambigua - esto quiere decir, que solamente un objeto en el repositorio actual comience con ese SHA-1 parcial.

Por ejemplo, para ver un “commit” específico, supongamos que se utiliza el comando `git log` y se identifica el “commit” donde se agregó cierta funcionalidad:

```
$ 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
```

En este caso, se escoge `1c002dd....`. Si utilizas `git show` en ese “commit”, los siguientes comandos son iguales (asumiendo que las versiones cortas no son ambiguas):

```
$ git show 1c002dd4b536e7479fe34593e72e6c6c1819e53b
$ git show 1c002dd4b536e7479f
$ git show 1c002d
```

Git puede descubrir una abreviación corta y única del valor SHA-1. Si añades `--abbrev-commit` al comando `git log`, el resultado utilizará los valores cortos pero manteniéndolos únicos; por default utiliza siete caracteres pero los hace más largos de ser necesario para mantener el SHA-1 sin ambigüedades:

```
$ git log --abbrev-commit --pretty=oneline
ca82a6d changed the version number
085bb3b removed unnecessary test code
a11bef0 first commit
```

Generalmente, de ocho a diez caracteres son más que suficientes para ser únicos en un proyecto.

Como un ejemplo, el kernel Linux, que es un proyecto bastante grande con alrededor de 450 mil “commits” y 3.6 millones de objetos, no tiene dos objetos cuyos SHA-1s se superpongan antes de los primeros 11 caracteres.

UNA BREVE NOTA RESPECTO A SHA-1

Mucha gente se preocupa de que en cierto momento, fruto del azar, tendrán dos objetos en su repositorio cuyos hash tendrán el mismo valor SHA-1. Pero, ¿entonces qué?

Si sucede que realizas un “commit” y el objeto tiene el mismo hash que un objeto previo en tu repositorio, Git verá el objeto previo en tu base de datos y asumirá que ya estaba escrito. Si intentas mirar el objeto otra vez, siempre tendrás la data del primer objeto.

Sin embargo, debes ser consciente de cuán ridículamente improbable es este escenario. El digest SHA-1 es de 20 bytes o 160 bits. El número de objetos aleatorios necesario para asegurar un 50% de probabilidades de una única colisión bordea el 2^{80} (la fórmula para determinar la probabilidad de colisión es $p = (n(n-1)/2) * (1/2^{160})$). 2^{80} es 1.2×10^{24} o 1 millón de millones de millones. Esto es 1,200 veces el número de granos de arena en la Tierra.

Aquí hay un ejemplo para darte una idea de lo que tomaría para conseguir una colisión de SHA-1. Si todos los 6.5 mil millones de humanos en la Tierra estuvieran programando, y cada segundo, cada uno produjera el código equivalente a toda la historia del kernel Linux (3.6 mil millones de objetos en Git) e hicieran push en un enorme repositorio Git, tomaría aproximadamente 2 años hasta que el repositorio tuviera suficientes objetos para un 50% de probabilidades de que ocurriera una única colisión en los SHA-1 de los objetos. Existe una probabilidad más alta de que cada miembro de tu equipo de programación sea atacado y asesinado por lobos en incidentes sin relación y todo esto en la misma noche.

NOTA

Referencias por rama

El camino más sencillo para especificar un “commit” requiere que este tenga una rama de referencia apuntando al mismo. Entonces, se puede usar el nombre de la rama en cualquier comando Git que espere un objeto “commit” o un valor SHA-1. Por ejemplo, si se quiere mostrar el último objeto “commit” de una rama, los siguientes comandos son equivalentes, asumiendo que la rama `topic1` apunta a `ca82a6d`:

```
$ git show ca82a6dff817ec66f44342007202690a93763949  
$ git show topic1
```

Si se quiere ver a qué SHA-1 apunta un rama en específico, o si se quiere ver lo que cualquiera de estos ejemplos expresa en términos de SHA-1s, puede utilizar una herramienta de plomería de Git llamada `rev-parse`. Se puede ver [Los entresijos internos de Git](#) para más información sobre las herramientas de plomería; básicamente, `rev-parse` existe para operaciones de bajo nivel y no está diseñado para ser utilizado en operaciones diarias. Aquí puedes correr `rev-parse` en tu rama.

```
$ git rev-parse topic1  
ca82a6dff817ec66f44342007202690a93763949
```

Nombres cortos de RefLog

Una de las cosas que Git hace en segundo plano, mientras tu estás trabajando a distancia, es mantener un “reflog” - un log de dónde se apuntan las referencias de tu HEAD y tu rama en los últimos meses.

Se puede ver el reflog utilizando `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
```

Cada vez que la punta de tu rama es actualizada por cualquier razón, Git guarda esa información en este historial temporal. Y es así como se puede especificar *commits* antiguos con esta información. Si se quiere ver el quinto valor anterior a tu HEAD en el repositorio, se puede usar la referencia `@{n}` que se ve en la salida de reflog:

```
$ git show HEAD@{5}
```

También se puede utilizar esta sintaxis para ver dónde se encontraba una rama dada una cierta cantidad de tiempo. Por ejemplo, para ver dónde se encontraba tu rama `master` ayer, se puede utilizar

```
$ git show master@{yesterday}
```

Esto muestra a dónde apuntaba tu rama el día de ayer. Esta técnica solo funciona para información que permanece en tu *reflog*, por lo que no se puede utilizar para ver *commits* que son anteriores a los que aparecen en él.

Para ver información sobre *reflog* en el formato de `git log`, se puede utilizar `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'

```

Es importante notar que la información de *reflog* es estrictamente local - es un log de lo que se ha hecho en el repositorio local. Las referencias no serán las mismas en otra copia del repositorio; y justo después de que se ha inicializado el repositorio, se tendrá un *reflog* vacío, dado que no ha ocurrido ninguna actividad todavía en el mismo. Utilizar `git show HEAD@{2.months.ago}` funcionará solo si se clonó el proyecto hace al menos dos meses - si se clonó hace cinco minutos, no se obtendrán resultados.

Referencias por ancestros

Otra forma principal de especificar un “commit” es por sus ancestros. Si se coloca un `^` al final de la referencia, Git lo resuelve como el padre de ese “commit”. Supongamos que se mira a la historia de un proyecto:

```

$ 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

```

Entonces, se puede ver los commits previos especificando `HEAD^`, lo que significa “el padre de 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'
```

También se puede especificar un número después de `^` - por ejemplo, `d921970^2` significa “el segundo padre de d921970.” Esta sintaxis es útil solamente para fusiones confirmadas, las cuales tienen más de un parente. El primer parente es la rama en el que se estaba al momento de fusionar, y el segundo es el “commit” en la rama en la que se fusionó:

```
$ 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
```

La otra manera principal de especificar ancestros es el `~`. Este también refiere al primer parente, así que `HEAD~` y `HEAD^` son equivalentes. La diferencia se vuelve aparente cuando se especifica un número. `HEAD~2` significa “el primer parente del primer parente,” o “el abuelo” - este recorre el primer parente las veces que se especifiquen. Por ejemplo, en el historial listado antes, `HEAD~3` sería

```
$ git show HEAD~3  
commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d  
Author: Tom Preston-Werner <tom@mojombo.com>  
Date: Fri Nov 7 13:47:59 2008 -0500  
  
    ignore *.gem
```

Esto también puede ser escrito `HEAD^{3}`, lo que también es, el primer parente del primer parente del primer parente:

```
$ git show HEAD^^^
commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d
Author: Tom Preston-Werner <tom@mojombo.com>
Date:   Fri Nov 7 13:47:59 2008 -0500

    ignore *.gem
```

También se puede combinar estas sintaxis - se puede obtener el segundo parente de la referencia previa (asumiendo que fue una fusión confirmada) utilizando `HEAD~3^2`, y así sucesivamente.

Rangos de Commits

Ahora que ya puede especificar *commits* individuales, vamos a ver cómo especificar un rango de *commits*. Esto es particularmente útil para administrar las ramas - si se tienen muchas ramas, se puede usar un rango de especificaciones para contestar preguntas como, “¿Qué trabajo está en esta rama y cuál no hemos fusionado en la rama principal?”

Dos puntos

La forma más común de especificar un rango es mediante la sintaxis de doble punto. Esto básicamente pide a Git que resuelva un rango de *commits* que es alcanzable desde un “commit” pero que no es alcanzable desde otro. Por ejemplo, digamos que se tiene un historial de *commits* que se ve como [Example history for range selection..](#)

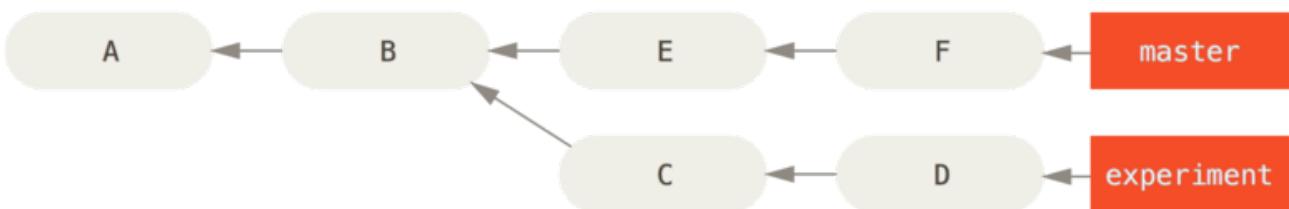


Figura 137. Example history for range selection.

Se quiere ver qué se encuentra en la rama experiment que no ha sido fusionado a la rama master todavía. Se puede pedir a Git que muestre el log de solamente aquellos *commits* con `master..experiment` - eso significa “todos los commits alcanzables por experiment que no son alcanzables por master.” Para ser breves y claros en este ejemplo. Se usarán las letras de los objetos “commit” del diagrama en lugar del log para que se muestre de la siguiente manera:

```
$ git log master..experiment
D
C
```

Si, por otro lado, se quiere lo opuesto - todos los commits en `master` que no están en `experiment` - se pueden invertir los nombres de las ramas. `experiment..master`` muestra todo

lo que hay en `master` que no es alcanzable para `experiment`:

```
$ git log experiment..master
F
E
```

Esto es útil si se quiere mantener la rama `experiment` actualizada y previsualizar lo que se está a punto de fusionar. Otro uso bastante frecuente de esta sintaxis es para ver lo que se está a punto de publicar en remote:

```
$ git log origin/master..HEAD
```

Este comando muestra cualquier “commit” en tu rama actual que no está en la rama `master` del remoto `origin`. Si se corre un `git push` y la rama de seguimiento actual es `origin/master`, los *commits* listados por `git log origin/master..HEAD` son los *commits* que serán transferidos al servidor. También se puede dejar de lado la sintaxis para que Git asuma la `HEAD`. Por ejemplo, se puede obtener el mismo resultado que en el ejemplo previo tipiando `git log origin/master..` - Git sustituye `HEAD` si un lado está faltando.

Múltiples puntos

La sintaxis de dos puntos es útil como una abreviatura; pero tal vez se desea especificar más de dos ramas para indicar la revisión, como puede ser revisar qué *commits* existen en muchas ramas que no se encuentran en la rama en la que se realiza el trabajo actualmente. Git permite realizar esto utilizando el carácter `^` o el comando `--not` antes de cualquier referencia de la cual no deseas ver los *commits* alcanzables.

```
$ git log refA..refB
$ git log ^refA refB
$ git log refB --not refA
```

Esto es bueno porque con esta sintaxis se puede especificar más de dos referencias en una consulta, lo que no se puede hacer con la sintaxis de dos puntos. Por ejemplo, si se quiere ver todos los *commits* que son alcanzables desde `refA` o `refB` pero no desde `refC`, se puede escribir lo siguiente:

```
$ git log refA refB ^refC
$ git log refA refB --not refC
```

Esto lo convierte en un sistema de consultas muy poderoso que debería ayudar a descubrir qué hay en tus ramas.

Tres puntos

La última sintaxis de selección de rangos es la de tres puntos, que especifica todos los *commits* que son alcanzables por alguna de dos referencias, pero no por las dos al mismo tiempo. Mira atrás al ejemplo de historial de commits en [Example history for range selection..](#) Si se quiere ver lo que está en `master` o `experiment` pero no en ambos, se puede utilizar

```
$ git log master...experiment  
F  
E  
D  
C
```

Nuevamente, esto entrega la salida normal del `log` pero muestra solo la información de esos cuatro *commits*, apareciendo en el tradicional ordenamiento por fecha de “commit”.

Un cambio común para utilizar con el comando `log`, en este caso, es `--left-right`, el cual muestra en qué lado del rango se encuentra cada “commit”. Esto ayuda a hacer la información más útil:

```
$ git log --left-right master...experiment  
< F  
< E  
> D  
> C
```

Con estas herramientas, se puede hacer saber más fácilmente a Git qué “commit” o *commits* desea inspeccionar.

Organización interactiva

Git viene con unos cuantos scripts que hacen que algunas líneas de comando sean más fáciles de usar. Aquí, verás unos cuantos comandos interactivos que te ayudaran a preparar tus confirmaciones para incluir sólo ciertas combinaciones y partes de los archivos. Estás herramientas serán muy útiles si modificas unos cuantos archivos y decides que esos cambios estén en varias confirmaciones enfocadas, en lugar de en una gran confirmación problemática. De esta manera, puedes asegurarte de que tus confirmaciones sean conjuntos de cambios lógicamente separados y que puedan ser revisados fácilmente por los desarrolladores que trabajan contigo. Si empiezas `git add` con el `-i` o la opción `--interactive`, Git entra en un modo de celda interactiva, mostrando algo como esto:

```

$ 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>

```

Puedes ver que este comando te muestra una muy diferente vista de tu área de ensayo – básicamente la misma información que con `git status`, pero un poco más sucinto e informativo. Muestra los cambios que has realizado en la izquierda y cambios que no has hecho a la derecha.

Después de esto viene una sección de comandos. Aquí puedes ver un sin número de cosas, incluidos los archivos organizados, archivos sin organizar, partes de archivos organizados, agregar archivos sin seguimiento y ver las diferencias de lo que se ha modificado.

Organizar y desorganizar archivos

Si tecleas `2` o `u` en el `What now>` rápidamente, la secuencia de comandos solicita los archivos que deseas representar:

```

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>>

```

Para organizar los archivos de `TODO` e `index.html`, puedes teclear los números:

```

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>>

```

El `*` antes de cada archivo significa que el archivo fue seleccionado para ser organizado. Si presionas `Enter` después de no escribir nada en el `Update>>` rápidamente, Git toma cualquier cosa seleccionada y la organiza por ti:

```

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

```

Ahora puedes ver que los archivos de TODO e index.html han sido organizados y el archivo simplegit.rb aún está sin organizar. Si deseas quitar el archivo TODO en este punto, use la opción **3** o **r** (para revertir):

```

*** 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

```

Mirando el estatus de tu Git de nuevo, puedes ver que has desordenado el archivo TODO:

```

*** 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

```

Para ver la diferencia de lo que ya has ordenado, puedes usar el comando **6** o **d** (para diferente). Éste te muestra una lista de tus archivos organizados, y puedes seleccionar aquellos de los que quisieras ver la diferencia de su organización. Esto es como

especificar el `git diff --cached` en la línea de comando:

```
*** 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">
```

Con estos comandos básicos, puedes usar el modo de añadir interactivo para tratar con tu área de organización un poco más fácilmente.

Parches de organización

De igual manera es posible para Git, el organizar ciertas partes de archivos y no todos los demás. Por ejemplo, si haces dos simples cambios en tu archivo simplegit.rb y quieres organizar uno pero no el otro, hacer esto es muy fácil en Git. Desde el prompt interactivo, teclea `5` o `p` (para parche). Git te preguntará qué archivos quieras organizar parcialmente; entonces, para cada sección de los archivos seleccionados, mostrará bloques del archivo diferencial y te preguntará si quisieras organizarlos uno por uno:

```

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,?]?

```

Tienes muchas opciones en este punto. Teclear `?` te mostrará una lista de lo que puedes hacer:

```

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

```

Generalmente, teclearías `y` o `n`, si quisieras organizar cada bloque, pero organizar cada uno de ellos en ciertos archivos o saltarte una decisión para algún bloque puede ser de ayuda para más tarde también. Si organizas una parte del archivo y dejas la otra parte sin organizar, su salida de estado se verá así:

```

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

```

El estatus del archivo `simplegit.rb` es interesante. Te muestra que un par de líneas están organizadas y otro par está desorganizado. Has organizado parcialmente este archivo. En este punto, puedes salir del script de adición interactivo y ejecutar `git commit` para

confirmar los archivos parcialmente organizados.

De igual manera, no necesitas estar en el modo de adición interactiva para hacer la organización parcial de archivos. Puedes iniciar el mismo script usando `git add -p` o `git add --patch` en la línea de comando.

Además, puede usar el modo de parche para restablecer parcialmente los archivos con el comando `reset --patch`, para verificar partes de archivos con el comando `checkout --patch` y para esconder partes de archivos con el comando `stash save --patch`. Vamos a entrar en más detalles sobre cada uno de éstos a medida que accedemos a usos más avanzados de dichos comandos.

Guardado rápido y Limpieza

Muchas veces, cuando has estado trabajando en una parte de tu proyecto, las cosas se encuentran desordenadas y quieres cambiar de ramas por un momento para trabajar en algo más. El problema es que no quieres hacer un “commit” de un trabajo que va por la mitad, así puedes volver a ese punto más tarde. La respuesta a ese problema es el comando `git stash`.

El guardado rápido toma el desorden de tu directorio de trabajo – que es, tus archivos controlados por la versión modificados y cambios almacenados – y lo guarda en un saco de cambios sin terminar que puedes volver a usar en cualquier momento.

Guardando rápido tu trabajo

Para demostrarlo, irás a tu proyecto y empezarás a trabajar en un par de archivos y posiblemente en tu etapa uno de cambios. Si ejecutas `git status`, puedes ver tu estado sucio:

```
$ 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
```

Ahora quieres cambiar de rama, pero no quieres hacer “commit”, todavía, a lo que has estado trabajando; así que le harás un guardado rápido a los cambios. Para poner un nuevo guardado rápido en tus archivos, ejecuta `git stash` o `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")
```

Tu directorio de trabajo está limpio:

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

En este punto, puedes fácilmente cambiar de ramas y hacer trabajos en otro lugar; tus cambios están guardados en tus archivos. Para ver qué guardados rápidos has almacenado, puedes usar `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
```

En este caso, dos guardados fueron hechos previamente, así que tienes tres diferentes trabajos guardados. Puedes volver a aplicar el que acabas de guardar utilizando el comando que se muestra en la salida de ayuda del comando original: `git stash apply`. Si quieres hacer entrada a uno de los guardados rápidos anteriores, puedes especificarlo poniendo su nombre de esta manera: `git stash apply stash@{2}`. Si no especificas un guardado, Git adopta el guardado más reciente e intenta hacerle entrada:

```
$ 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
#
```

Puedes ver que Git remodifica los archivos que revertiste cuando hiciste el guardado rápido. En este caso, tenías un directorio de trabajo despejado cuando intentaste hacer entrada al guardado, e intentaste hacerle entrada en la misma rama en la que lo guardaste; pero tener un directorio de trabajo despejado y usarlo en la misma rama no es necesario para hacerle entrada a un guardado con éxito. Puedes almacenar un guardado en una rama, cambiar a otra rama luego, e intentar volver a hacerle entrada a los cambios. También puedes tener archivos modificados y sin aplicar en tu directorio de trabajo cuando des entrada a un guardado – Git te da conflictos de combinación

si algo ya no se usa de manera limpia.

Los cambios a tus archivos fueron reaplicados, pero el archivo que tu guardaste antes no fue realmacenado. Para hacer eso, tienes que ejecuar el comando `git stash apply` con una opción de `--index` para decirle al comando que intente reaplicar los cambios almacenados. Si anteriormente lo hubieras ejecutado, lo habrías vuelto a tener en su posición original:

```
$ 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
#
```

La opción de hacer entrada sólo intenta hacer entrada al trabajo guardado –lo continúas teniendo en tus archivos. Para removerlo, puedes ejecutar `git stash drop` con el nombre del guardado a eliminar:

```
$ 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)
```

También puedes ejecutar `git stash pop` para hacer entrada al guardado y luego eliminarlo inmediatamente de tus archivos.

Guardado rápido creativo

Hay algunas pocas variantes de guardado rápido que pueden ser útiles también. La primera opción que es muy popular es `--keep-index` para el comando `stash save`. Esto le dice a Git que no guarde nada que tú ya hayas almacenado con el comando `git add`.

Esto puede ser útil de verdad si has hecho un buen número de cambios, pero sólo quieres aplicar permanentemente algunos de ellos y luego regresar al resto de cambios en una siguiente ocasión.

```
$ 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
```

Otra cosa común que puede que quieras hacer con tus guardados es hacer un guardado rápido de los archivos que no están bajo control de la versión al igual que con los que lo están. Por defecto, `git stash` solamente guardará archivos que ya están en el índice. Si especificas `--include-untracked` o `-u`, Git también hará un guardado rápido de cualquier archivo que no esté bajo control de la versión que hayas creado.

```
$ 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
$
```

Finalmente, si especificas la flag `--patch`, Git no hará guardado rápido de todo lo que es modificado, pero, en su lugar, te recordará cuales de los cambios te gustaría guardar y cuales te gustaría mantener en tu trabajo directamente.

```

$ 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,?]? y

Saved working directory and index state WIP on master: 1b65b17 added the index file

```

Creando una Rama desde un Guardado Rápido

Si haces guardado rápido de algo de trabajo, lo dejas ahí por un rato y continúas en la rama de la cual hiciste guardado rápido de tu trabajo, puede que tengas problemas rehaciendo la entrada al trabajo. Si la entrada intenta modificar un archivo que desde entonces has modificado, tendrás un conflicto de combinación y tendrás que intentar resolverlo. Si quieres una forma más fácil de probar los cambios guardados de nuevo, puedes ejecutar `git stash branch`, el cual crea una nueva rama para ti, verifica el “commit” en el que estabas cuando hiciste guardado rápido de tu trabajo, recrea tu trabajo allí, y luego arroja el guardado rápido si la entrada se realiza con éxito:

```

$ 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)

```

Este es un buen método rápido para recuperar trabajos guardados y trabajar en una nueva rama.

Limpiendo tu Directorio de Trabajo

Finalmente, puede que no quieras hacer guardado rápido de algo de trabajo o de archivos en tu directorio de trabajo, pero quieres deshacerte de ellos. El comando `git clean` hará esto por ti.

Algunas razones comunes para esto pueden ser: remover archivos cruft que han sido generados por herramientas de combinación o externas, o para eliminar viejos archivos de versión con el fin de ejecutar una versión limpia.

Querrás ser más bien delicado con este comando, ya que está diseñado para eliminar archivos de tu directorio de trabajo que no están siendo tomados en cuenta. Si cambias de opinión, muchas veces no hay restauración para el contenido de esos archivos. Una opción más segura es ejecutar `git stash --all` para eliminar todo, pero lo almacena en un guardado.

Asumiendo que quieres eliminar los archivos cruft o limpiar tu directorio de trabajo, puedes hacerlo con `git clean`. Para remover los archivos que no están bajo el control de la versión en tu directorio de trabajo, puedes ejecutar `git clean -f -d`, el cual remueve cualquier archivo y también cualquier subdirectorio que se vuelva vacío como resultado. El `-f` significa *fuerza* o “realmente haz esto”.

Si alguna vez quieres ver que haría, puedes ejecutar el comando con la opción `-n` que significa ‘`haz un arranque en seco y dime que habrías eliminado`’.

```
$ git clean -d -n
Would remove test.o
Would remove tmp/
```

Por defecto, el comando `git clean` sólo removerá archivos que no sean controlados y que no sean ignorados. Cualquier archivo que empareje en patrón en tu `.gitignore` u otros archivos ignorados no serán removidos. Si quieres eliminar esos archivos también, como eliminar todos los `.o` generados por la versión, así puedes hacer una versión completamente limpia, puedes añadir un `-x` al comando.

```
$ 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/
```

Si no sabes lo que el comando `git clean` va a hacer, siempre ejecútalo con un `-n` primero para estar seguro antes de cambiar el `-n` a `-f` y hacerlo de verdad. La otra forma en la que puedes ser cuidadoso con el proceso es ejecutarlo con el `-i` o con la flag “interactive”.

Esto ejecutará el comando en limpio en un modo interactivo.

```
$ 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
each          5: quit
  6: help
What now>
```

De esta forma puedes decidir por cada archivo individualmente o especificar los términos para la eliminación de forma interactiva.

Firmando tu trabajo

Git es criptográficamente seguro, pero no es a prueba de tontos. Si estás tomando trabajo de otros de Internet y quieres verificar que los *commits* son realmente de fuentes seguras, Git tiene unas cuantas maneras de firmar y verificar utilizando GPG.

Introducción a GPG

Antes que nada, si quieres firmar cualquier cosa necesitas tener configurado GPG y tu llave personal instalada.

```
$ 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
```

Si no tienes una llave instalada, puedes generar una con `gpg --gen-key`.

```
gpg --gen-key
```

Una vez que tengas una llave privada para firmar, puedes configurar Git para usarla y firmar cosas configurando la opción `user.signingkey`.

```
git config --global user.signingkey 0A46826A
```

Ahora Git usará tu llave por defecto para firmar *tags* y *commits* si tuquieres.

Firmando Tags

Si tienes una llave GPG privada configurada, ahora puedes usarla para firmar *tags*. Todo lo que tienes que hacer es usar `-s` en lugar de `-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
```

Si ejecutas `git show` en ese *tag*, puedes ver tu firma GPG adjunta a él:

```
$ 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

iQEcBAABAgAGBQJTzbQIAoJEF0+sviABDDrZbQH/09PfE51KPVPlanr6q1v4/Ut
LQxfojUWiLQdg2ESJItkcuweYg+kc3HCyFejeDIBw9dpXt00rY26p05qrpnG+85b
hM1/PswpPLuBSr+oCIDj5GMC2r2iEKsfv2fJbNW8iWAXVLoWZRF8B0MfqX/YTMbm
ecorc4iXzQu7tupRihslbNkfvcimnSDeSvzCpWAH17h8Wj6hhqePmLm9lAYqnKp
8S5B/1SSQuEAjRZgI4IexpZoeKGVDptPHxLLS38fozsyi0QyDyzEgJxcJQVMXxVi
RUysgqjcpT8+iQM1Pb1GfHR4XAh0qN5Fx06PSaFZhqvWFezJ28/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
```

Verificando Tags

Para verificar un tag firmado, usa `git tag -v [nombre-de-tag]`. Este comando usa GPG para verificar la firma. Necesitas tener guardada la llave pública del usuario para que esto funcione de manera apropiada:

```
$ 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

Si no tienes la llave pública de quien firmó, obtendrás algo como esto en cambio:

```
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'
```

Firmando Commits

En versiones más recientes de Git (v1.7.9 en adelante), ahora puedes firmar *commits* individuales. Si estás interesado en firmar *commits* directamente en lugar de solo los *tags*, todo lo que necesitas hacer es agregar un **-S** a tu comando `git commit`.

```
$ 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
```

Para ver y verificar las firmas, también existe una opción `--show-signature` para `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
```

Adicionalmente, puedes configurar `git log` para verificar cualquier firma que encuentre y listarlas en su salida con el formato `%G?`.

```
$ 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
```

Aquí podemos ver que sólo el último *commit* es firmado y válido y los *commits* previos no.

En Git 1.8.3 y posteriores, "git merge" y "git pull" pueden ser configurados para inspeccionar y rechazar cualquier *commit* que no adjunte una firma GPG de confianza con el comando `--verify-signatures`.

Si se usa esta opción cuando se fusiona una rama y esta contiene *commits* que no están firmados, aunque sean válidos, la fusión no funcionará.

```
$ git merge --verify-signatures non-verify
fatal: Commit ab06180 does not have a GPG signature.
```

Si una fusión contiene solo *commits* válidos y firmados, el comando `merge` mostrará todas las firmas que ha revisado y después procederá con la fusión.

```
$ 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(+)
```

También se puede utilizar la opción `-S` junto con el mismo comando `git merge` para firmar el *commit* resultante. El siguiente ejemplo verifica que cada *commit* en la rama por ser fusionada esté firmado y también firma el *commit* resultado de la fusión.

```
$ 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(+)
```

Todos deben firmar

Firmar *tags* y *commits* es grandioso, pero si decides usar esto en tu flujo de trabajo normal, tendrás que asegurarte que todos en el equipo entiendan cómo hacerlo. Si no, terminarás gastando mucho tiempo ayudando a las personas a descubrir cómo reescribir sus *commits* con versiones firmadas. Asegúrate de entender GPG y los beneficios de firmar cosas antes de adoptarlo como parte de tu flujo de trabajo normal.

Buscando

Con casi cualquier tamaño de código de base, a menudo necesitará encontrar en dónde se llama o define una función, o encontrar el historial de un método. Git proporciona un par de herramientas útiles para examinar el código y hacer *commit* a las instantáneas almacenadas en su base de datos de forma rápida y fácil. Vamos a revisar algunas de ellas.

Git Grep

Git se envía con un comando llamado `grep` que le permite buscar fácilmente a través de cualquier árbol o directorio de trabajo con *commit* por una cadena o expresión regular. Para estos ejemplos, veremos el código fuente de Git.

Por defecto, mirará a través de los archivos en su directorio de trabajo. Puedes pasar `-n` para imprimir los números de línea donde Git ha encontrado coincidencias.

```
$ 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
```

Hay una serie de opciones interesantes que puede proporcionar el comando `grep`.

Por ejemplo, en lugar de la llamada anterior, puedes hacer que Git resuma el resultado simplemente mostrándote qué archivos coinciden y cuántas coincidencias hay en cada archivo con la opción `--count`:

```
$ 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
```

Si quieres ver en qué método o función piensa Git que ha encontrado una coincidencia, puedes pasar `-P`:

```
$ git grep -P gmtime_r *.c
date.c=static int match_multi_number(unsigned long num, char c, const char *date, char
*end, struct tm *tm)
date.c:            if (gmtime_r(&now, &now_tm))
date.c=static int match_digit(const char *date, struct tm *tm, int *offset, int
*tm_gmt)
date.c:            if (gmtime_r(&time, tm)) {
```

Así que aquí podemos ver que se llama a `gmtime_r` en las funciones `match_multi_number` y `match_digit` en el archivo `date.c`.

También puedes buscar combinaciones complejas de cadenas con el indicador `--and`, que asegura que múltiples coincidencias estén en la misma línea. Por ejemplo, busquemos cualquier línea que defina una constante con las cadenas “LINK” o “BUF_MAX” en ellas en la base del código de Git en una versión 1.8.0 anterior.

Aquí también usaremos las opciones `--break` y `--heading` que ayudan a dividir el resultado en un formato más legible.

```
$ 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 */
```

El comando `git grep` tiene algunas ventajas sobre los comandos de búsqueda normales como `grep` y `ack`. La primera es que es realmente rápido, la segunda es que puedes buscar a través de cualquier árbol en Git, no solo en el directorio de trabajo. Como vimos en el ejemplo anterior, buscamos términos en una versión anterior del código fuente de Git, no en la versión que estaba actualmente verificada.

Búsqueda de Registro de Git

Quizás no estás buscando **dónde** existe un término, sino **cuándo** existió o se introdujo. El comando `git log` tiene varias herramientas potentes para encontrar *commits* específicos por el contenido de sus mensajes o incluso el contenido de las diferencias que introducen.

Si queremos saber, por ejemplo, cuándo se introdujo originalmente la constante `ZLIB_BUF_MAX`, podemos decirle a Git que sólo nos muestre los *commits* que agregaron o eliminaron esa cadena con la opción `-S`.

```
$ git log -SZLIB_BUF_MAX --oneline
e01503b zlib: allow feeding more than 4GB in one go
ef49a7a zlib: zlib can only process 4GB at a time
```

Si miramos la diferencia de esos *commits*, podemos ver que en `ef49a7a` se introdujo la constante y en `e01503b` se modificó.

Si necesitas ser más específico, puedes proporcionar una expresión regular para buscar con la opción `-G`.

Búsqueda de Registro de Línea

Otra búsqueda de registro bastante avanzada que es increíblemente útil es la búsqueda del historial de línea. Esta es una adición bastante reciente y no muy conocida, pero puede ser realmente útil. Se llama con la opción `-L` a `git log` y te mostrará el historial de una función o línea de código en tu base de código.

Por ejemplo, si quisieramos ver cada cambio realizado en la función `git_deflate_bound` en el archivo `zlib.c`, podríamos ejecutar `git log -L :git_deflate_bound:zlib.c`. Esto intentará descubrir cuáles son los límites de esa función y luego examinará el historial y nos mostrará cada cambio que se hizo a la función como una serie de parches de cuando se creó la función por primera vez.

```
$ 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);
+}
+
```

Si Git no puede encontrar la forma de relacionar una función o método en tu lenguaje de programación, también puedes proporcionarle una expresión regular. Por

ejemplo, esto habría hecho lo mismo: `git log -L '/unsigned long git_deflate_bound',/^}:/zlib.c`. También podrías darle un rango de líneas o un único número de línea y obtendrás el mismo tipo de salida.

Reescribiendo la Historia

Muchas veces, al trabajar con Git, vas a querer confirmar tu historia por alguna razón. Una de las grandes cualidades de Git es que te permite tomar decisiones en el último momento. Puede decidir qué archivos entran en juego antes de comprometerse con el área de ensayo, puedes decidir que no querías estar trabajando en algo todavía con el comando de alias, y puedes reescribir confirmaciones que ya hayan pasado haciendo parecer que fueron hechas de diferente manera. Esto puede desenvolverse en el cambio de las confirmaciones, cambiando mensajes o modificando los archivos en un cometido, aplastando o dividiendo confirmaciones enteramente – todo antes de que compartas tu trabajo con otras personas.

En esta sección, verás cómo complementar esas tareas tan útiles que harán a la confirmación de tu historia aparecer del modo en el cual quisiste compartirla.

Cambiando la última confirmación

Cambiar la última confirmación es probablemente lo más común que le harás a tu historia. Comúnmente querrás hacer dos cosas en tu última confirmación: cambiar la confirmación del mensaje, o cambiar la parte instantánea que acabas de agregar sumando, cambiando y/o removiendo archivos.

Si solamente quieres cambiar la confirmación del mensaje final, es muy sencillo:

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

Esto te envía al editor de texto, el cual tiene tu confirmación final, listo para modificarse en el mensaje. Cuando guardes y cierres el editor, el editor escribe una nueva confirmación conteniendo el mensaje y lo asigna a tu última confirmación.

Si ya has cambiado tu última confirmación y luego quieres cambiar la instantánea que confirmaste al agregar o cambiar archivos, porque posiblemente olvidaste agregar un archivo recién creado cuando se confirmó originalmente, el proceso trabaja prácticamente de la misma manera. Tu manejas los cambios que quieras editando el archivo y oprimiendo `git add` en éste o `git rm` a un archivo adjunto, y el subsecuente `git commit --amend` toma tu área de trabajo actual y la vuelve una instantánea para la nueva confirmación.

Debes ser cuidadoso con esta técnica porque puedes modificar los cambios del SHA-1 de la confirmación. Es como un muy pequeño *rebase* – no necesitas modificar tu última confirmación si ya lo has hecho.

Cambiando la confirmación de múltiples mensajes

Para modificar una confirmación que está más atrás en tu historia, deberás aplicar herramientas más complejas Git no tiene una herramienta para modificar la historia, pero puedes usar la herramienta de *rebase* para rebasar ciertas series de confirmaciones en el HEAD en el que se basaron originalmente en lugar de moverlas a otro. Con la herramienta interactiva del *rebase*, puedes parar justo después de cada confirmación que quieras modificar y cambiar su mensaje, añadir archivos, o hacer cualquier cosa que quieras. Puedes ejecutar el *rebase* interactivamente agregando el comando `-i` a `git rebase`. De igual manera debes indicar que tan atrás quieras regresar para reescribir las confirmaciones escribiendo en el comando cuál confirmación quieras rebasar.

Por ejemplo, si quieras cambiar las confirmaciones de los tres últimos mensajes, o cualquiera de los mensajes de confirmación de ese grupo, proporcionas un argumento para el `git rebase -i` que quieras modificar de tu última confirmación, el cual es `HEAD~2^` o `HEAD~3`. Debería ser más fácil el recordar el `~3` porque estás tratando de editar las últimas tres confirmaciones; pero ten en mente que estás designando actualmente cuatro confirmaciones atrás, aparte del último cometido que deseas editar:

```
$ git rebase -i HEAD~3
```

Recuerda de Nuevo que este es un comando de *rebase* – cualquier confirmación incluida en el rango de `HEAD~3..HEAD` será reescrita, aún si cambias el mensaje o no. No incluyas cualquier confirmación que ya hayas enviado al servidor central – si lo haces esto confundirá a los demás desarrolladores proporcionando una versión alternativa del mismo cambio.

Utilizar este comando te da una lista de las confirmaciones en tu editor de texto que se ve como este:

```

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

```

Es importante el notar que estas confirmaciones son escuchadas en el orden contrario del que tú normalmente las verías usando el comando de `log`. Si utilizaras un comando de `log`, verías algo como esto.

```
$ 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
```

Nótese que el orden está al revés. El *rebase* interactivo te da un script que va a utilizarse. Este empezará en la confirmación que específicas en la línea de comandos (`HEAD~3`) y reproducirá los cambios introducidos en cada una de estas confirmaciones de arriba a abajo. Este acomoda los más viejos en la parte de arriba, y va bajando hasta los más nuevos, porque ese será el primero en reproducirse

Necesitarás editar el script para que se detenga en la confirmación que quieras editar. Para hacer eso, cambia la palabra `pick` por la frase `edit` para cada una de las confirmaciones en las que quieras que el script se detenga. Por ejemplo, para modificar solamente la tercera confirmación del mensaje, cambiarías el archivo para que se viera algo así:

```
edit f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

Cuando guardes y salgas del editor, Git te enviará atrás a la última confirmación en la lisa y te llevará a la línea de comando con el siguiente mensaje:

```
$ 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
```

Estas instrucciones te dirán exactamente qué hacer. Type

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

Cambia la confirmación del mensaje, y sal del editor. Then, run

```
$ git rebase --continue
```

Este comando te permitirá aplicar las otras dos confirmaciones automáticamente, y después de esto estás listo. Si decides cambiar y elegir editar en más líneas, puedes repetir estos pasos para cada confirmación que cambies en cada edición. Cada vez, Git se parará, permitiéndote modificar la confirmación y continuar cuando hayas terminado

Reordenando Confirmaciones

De igual manera puedes usar rebases interactivos para reordenar o remover confirmaciones enteramente. Si quieres remover la “added cat-file” confirmación y cambiar el orden en el cual las otras dos confirmaciones son introducidas, puedes cambiar el *rebase* en el script de esto:

```
pick f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

A esto:

```
pick 310154e updated README formatting and added blame
pick f7f3f6d changed my name a bit
```

Cuando guardes y salgas del editor, Git recordará tu rama de padres de estas

confirmaciones, aplicando 310154e y después f7f3f6d, y después se parará. Cambias efectivamente el orden de esas confirmaciones y eliminas la “added cat-file” confirmación completamente.

Unir confirmaciones

También es posible el tomar series de confirmaciones y unirlas todas en una sola confirmación con la herramienta interactiva de *rebase*. El script pone las instrucciones en el mensaje de rebase:

```
#  
# 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
```

Si, en vez de “`pick`” o “`edit`”, específicas “`squash`”, Git aplica a ambos este cambio y los cambia directamente después y hace que las confirmaciones se unan. Entonces, si quieres convertir en una única confirmación estas tres confirmaciones, deberás hacer que el script se vea como esto:

```
pick f7f3f6d changed my name a bit  
squash 310154e updated README formatting and added blame  
squash a5f4a0d added cat-file
```

Cuando guardes y salgas del editor, Git aplicará a los tres el cambio y después te dirigirá en el editor para fusionar los tres mensajes de la confirmación:

```
# 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
```

Cuando guardes eso, tendrás una única confirmación que introducirá los cambios de las tres previas confirmaciones.

Dividiendo una confirmación

Dividir una confirmación la deshace y después realiza etapas parciales de las confirmaciones tantas veces como confirmaciones deseas finalizar. Por ejemplo, suponiendo que quieras dividir la confirmación de en medio de tus tres confirmaciones. En vez de “updated README formatting and added blame”, quieres dividirla en dos confirmaciones: “updated README formatting” para la primera, y “added blame” para la segunda. Puedes hacer eso en el script `rebase -i` cambiando la instrucción en la confirmación que quieras dividir a “edit”:

```
pick f7f3f6d changed my name a bit  
edit 310154e updated README formatting and added blame  
pick a5f4a0d added cat-file
```

¿Entonces, cuando el script te envíe a la línea de comandos, tu reseteas esa confirmación, tomas los cambios que se han hecho, y creas múltiples confirmación fuera de ellas? Cuando guardes y salgas del editor, Git te enviará al padre de la primera confirmación en tu lista, aplicando a la primera confirmación (`f7f3f6d`), a la segunda (`310154e`) y te enviará directamente a la consola. Ahí, puedes hacer un reseteo mixto de esa confirmación con el `git reset HEAD^`, el que efectivamente deshace las confirmaciones en los archivos referidos. Ahora puedes organizar y confirmar los archivos hasta que tengas varias confirmaciones y ejecutar `git rebase --continue` cuando hayas terminado:

```
$ 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 aplica la última confirmación (`a5f4a0d`) en el script, y tu historia quedaría de

esta manera:

```
$ git log -4 --pretty=format:"%h %s"
1c002dd added cat-file
9b29157 added blame
35cfb2b updated README formatting
f3cc40e changed my name a bit
```

Una vez de Nuevo, esto cambia el SHA-1s de todas tus confirmaciones en tu lista, así que asegúrate de que ninguna confirmación esté en esa lista que ya has puesto en un repositorio compartido.

La opción nuclear: filtrar-ramificar

Existe otra opción en la parte de volver a escribir la historia que puedes usar si necesitas reescribir un gran número de confirmaciones de una manera que se puedan scriptear – de hecho, cambiar tu dirección de e-mail o remover cualquier archivo en las confirmaciones. El comando es `filter-branch`, y este puede reescribir una gran cantidad de franjas de tu historia, así que probablemente no lo deberías usar a menos que tu proyecto aún no sea público y otra persona no se haya basado en las confirmaciones que estás a punto de reescribir. Como sea, podría ser muy útil. Aprenderás unas cuantas maneras muy comunes de obtener una idea de algunas de las cosas que es capaz de hacer.

Remover un archivo de cada confirmación

Esto ocurre comúnmente. Alguien accidentalmente confirma un gran número binario de un archivo con un irreflexivo `git add .`, y quieres removerlo de todas partes. Suponiendo que accidentalmente confirmaste un archivo que contenía contraseña y quieres volverlo un proyecto abierto. `filter-branch` es la herramienta que tu probablemente quieras usar para limpiar toda tu historia. Para remover un archivo nombrado `passwords.txt` de tu historia complete puedes aplicar el comando `--tree-filter` a `filter-branch`:

```
$ git filter-branch --tree-filter 'rm -f passwords.txt' HEAD
Rewrite 6b9b3cf04e7c5686a9cb838c3f36a8cb6a0fc2bd (21/21)
Ref 'refs/heads/master' was rewritten
```

El `--tree-filter` inicia el comando específico después de cada revisión del proyecto y éste entonces vuelve a confirmar los resultados. En este caso, deberías remover el archivo llamado `passwords.txt` de cada instantánea, aún si existe o no. Si quieres remover todas las confirmaciones accidentales del respaldo del editor de archivos, puedes iniciar algo como el `git filter-branch --tree-filter 'rm -f *~' HEAD`.

Deberías ser capaz de ver la re-escripción de confirmaciones y estructuras de Git y luego debes mover el puntero de la rama al final. Es generalmente una buena idea hacer esto en una parte de prueba de la rama y hacer un *hard-reset* de tu rama principal después de haber determinado que el resultado es lo que realmente deseas.

Para iniciar `filter-branch` en todas las ramas, puedes poner `--all` en el comando.

Hacer que un subdirectorio sea la nueva raíz

Suponiendo que has hecho una importación desde otro centro de Sistema de Control y tienes subdirecciones que no tienen ningún sentido (tronco, etiquetas, etc). . Si quieres hacer que el subdirectorio `tronco` sea el nuevo proyecto de la raíz de cada confirmación, `filter-branch` te puede ayudar a hacer eso también:

```
$ git filter-branch --subdirectory-filter trunk HEAD
Rewrite 856f0bf61e41a27326cd8e8f09fe708d679f596f (12/12)
Ref 'refs/heads/master' was rewritten
```

Ahora la raíz de tu nuevo proyecto es la que solía estar en el subdirectorio `tronco` cada vez. Git automáticamente remueve confirmaciones que no afecten al subdirectorio.

Cambiar la dirección de e-mail globalmente

Otro caso común es que olvides iniciar el `git config` para poner tu nombre y tu dirección de e-mail antes de que hayas empezado a trabajar, o tal vez quieras abrir un proyecto en el trabajo y cambiar tu e-mail de trabajo por tu e-mail personal. En cualquier caso, puedes cambiar la dirección de e-mail de múltiples confirmaciones en un lote con `filter-branch` igual. Necesitas ser cuidadoso de cambiar sólo las direcciones de e-mail que son tuyas, de manera que debes usar `--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
```

Esto va a través de la re-escripción de cada confirmación para tener tu nueva dirección. Porque cada confirmación contiene el SHA-1 de sus padres, este comando cambia cada confirmación del SHA-1 en tu historia, no solamente aquellos en los cuales el e-mail es el mismo o encaja.

Reiniciar Desmitificado

Antes de pasar a herramientas más especializadas, hablemos de `reset` y `checkout`. Estos comandos son dos de las partes más confusas de Git cuando los encuentras por primera vez. Hacen tantas cosas, que parece imposible comprenderlas realmente y emplearlas adecuadamente. Para esto, recomendamos una metáfora simple.

Los Tres Árboles

Una manera más fácil de pensar sobre `reset` y `checkout` es a través del marco mental de Git como administrador de contenido de tres árboles diferentes. Por “árbol”, aquí realmente queremos decir “colección de archivos”, no específicamente la estructura de datos. (Hay algunos casos donde el índice no funciona exactamente como un árbol, pero para nuestros propósitos es más fácil pensarlo de esta manera por ahora).

Git como sistema maneja y manipula tres árboles en su operación normal:

| Árbol | Rol |
|-----------------------|--|
| HEAD | Última instantánea del commit, próximo padre |
| Índice | Siguiente instantánea del commit propuesta |
| Directorio de Trabajo | Caja de Arena |

El HEAD

HEAD es el puntero a la referencia de bifurcación actual, que es, a su vez, un puntero al último commit realizado en esa rama. Eso significa que HEAD será el padre del próximo commit que se cree. En general, es más simple pensar en HEAD como la instantánea de **tu último commit**.

De hecho, es bastante fácil ver cómo es el aspecto de esa instantánea. Aquí hay un ejemplo de cómo obtener la lista del directorio real y las sumas de comprobación SHA-1 para cada archivo en la instantánea de HEAD:

```
$ git cat-file -p HEAD
tree cfda3bf379e4f8dba8717dee55aab78aef7f4daf
author Scott Chacon 1301511835 -0700
committer Scott Chacon 1301511835 -0700

commit inicial

$ git ls-tree -r HEAD
100644 blob a906cb2a4a904a152... README
100644 blob 8f94139338f9404f2... Rakefile
040000 tree 99f1a6d12cb4b6f19... lib
```

Los comandos `cat-file` y `ls-tree` son comandos de “fontanería” que se usan para cosas de nivel inferior y que no se usan realmente en el trabajo diario, pero nos ayudan a ver qué está sucediendo aquí.

El Índice

El índice es tu **siguiente commit propuesto**. También nos hemos estado refiriendo a este concepto como el “Área de Preparación” de Git ya que esto es lo que Git ve cuando ejecutas `git commit`.

Git rellena este índice con una lista de todos los contenidos del archivo que fueron revisados por última vez en tu directorio de trabajo y cómo se veían cuando fueron revisados originalmente. A continuación, reemplaza algunos de esos archivos con nuevas versiones de ellos, y `git commit` los convierte en el árbol para un nuevo commit.

```
$ git ls-files -s
100644 a906cb2a4a904a152e80877d4088654daad0c859 0  README
100644 8f94139338f9404f26296befa88755fc2598c289 0  Rakefile
100644 47c6340d6459e05787f644c2447d2595f5d3a54b 0  lib/simplegit.rb
```

Nuevamente, aquí estamos usando `ls-files`, que es más un comando entre bastidores que te muestra a qué se parece actualmente el índice.

El índice no es técnicamente una estructura de árbol – en realidad se implementa como un manifiesto aplanado – pero para nuestros propósitos, está lo suficientemente cerca.

El Directorio de Trabajo

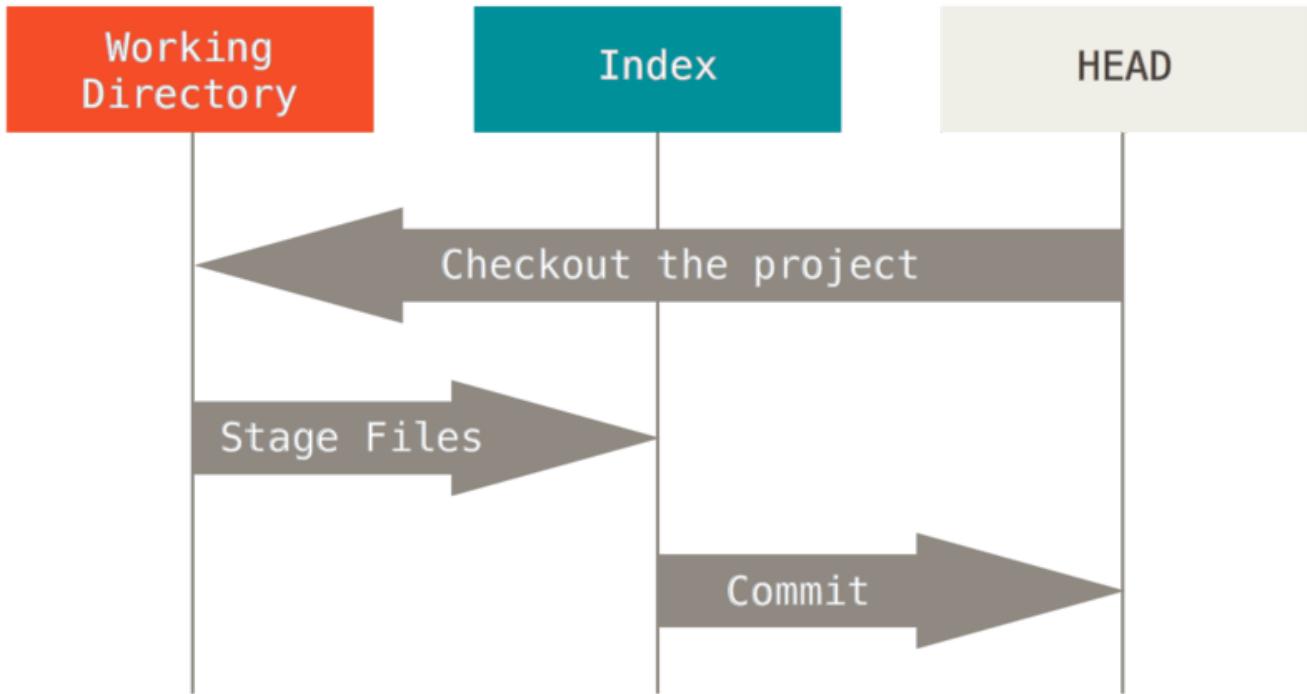
Finalmente, tienes tu directorio de trabajo. Los otros dos árboles almacenan su contenido de manera eficiente pero inconveniente, dentro de la carpeta `.git`. El Directorio de trabajo los descomprime en archivos reales, lo que hace que sea mucho más fácil para ti editarlos. Piensa en el Directorio de Trabajo como una **caja de arena**, donde puedes probar los cambios antes de enviarlos a tu área de ensayo (índice) y luego al historial.

```
$ tree
.
├── README
├── Rakefile
└── lib
    └── simplegit.rb

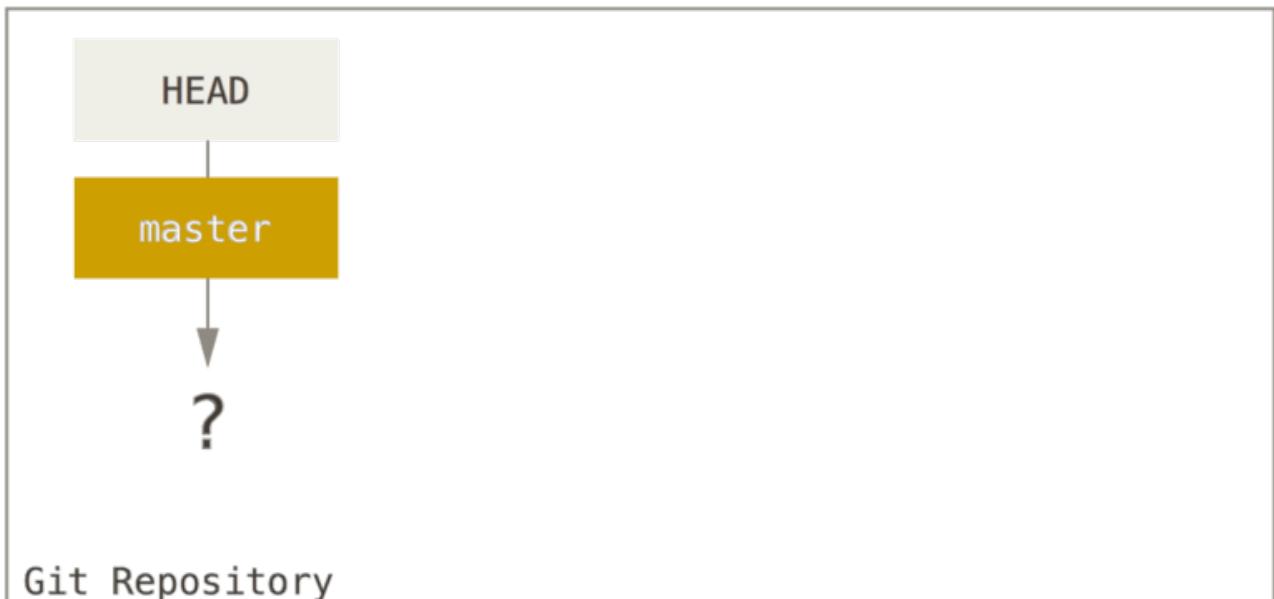
1 directory, 3 files
```

El Flujo de Trabajo

El objetivo principal de Git es registrar instantáneas de tu proyecto en estados sucesivamente mejores, mediante la manipulación de estos tres árboles.



Visualicemos este proceso: digamos que ingresa en un nuevo directorio con un solo archivo. Llamaremos a esto **v1** del archivo, y lo indicaremos en azul. Ahora ejecutamos `git init`, que creará un repositorio Git con una referencia HEAD que apunta a una rama no nacida (`master` aún no existe).

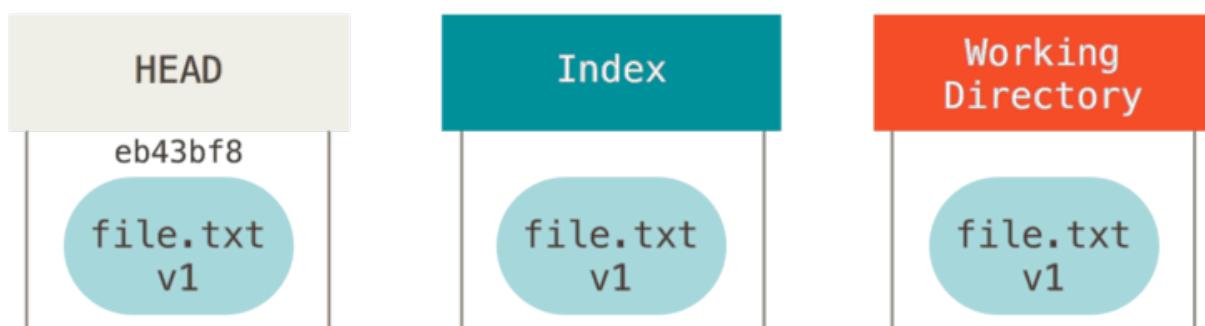
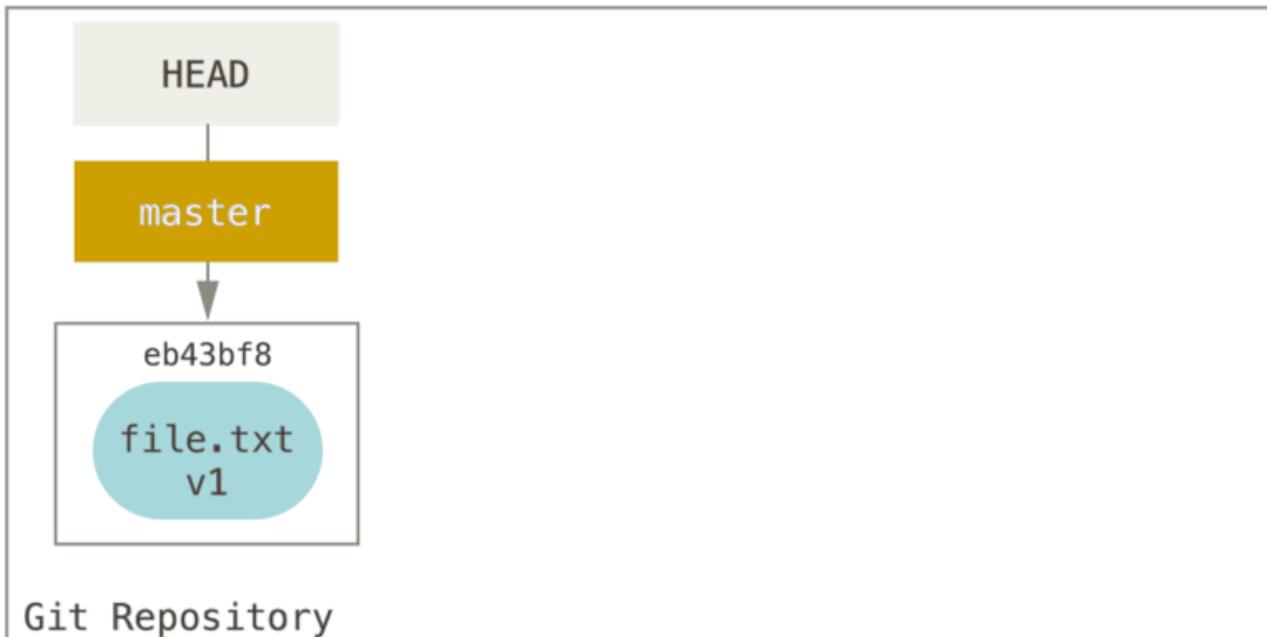


En este punto, solo el árbol del Directorio de Trabajo tiene cualquier contenido.

Ahora queremos hacer “commit” a este archivo, por lo que usamos `git add` para tomar contenido en el directorio de trabajo y copiarlo en el índice.



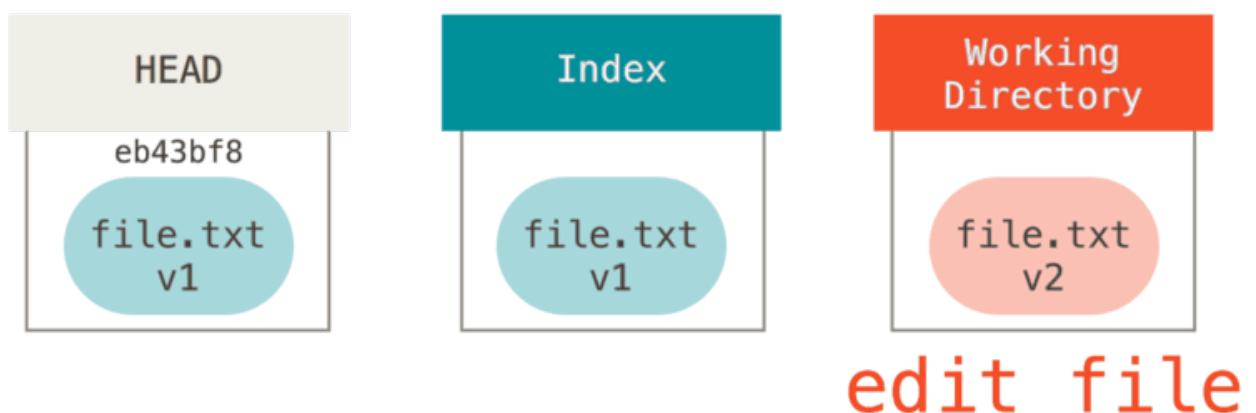
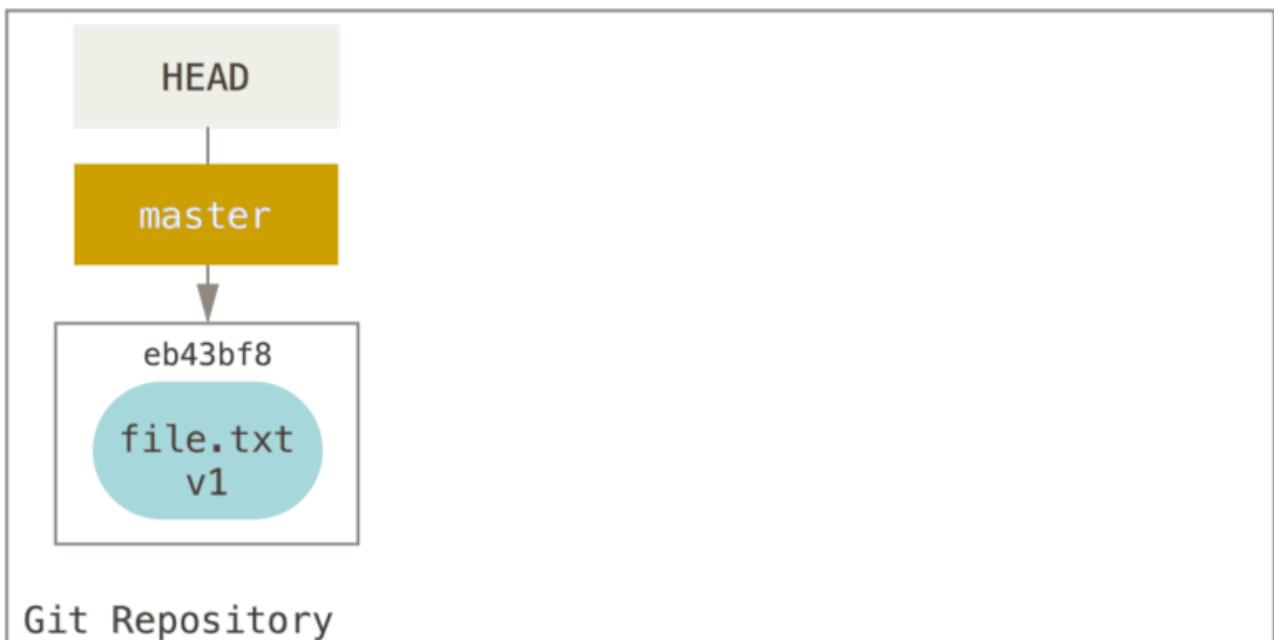
Luego ejecutamos `git commit`, que toma los contenidos del índice y los guarda como una instantánea permanente, crea un objeto de “commit” que apunta a esa instantánea y actualiza `master` para apuntar a ese “commit”.



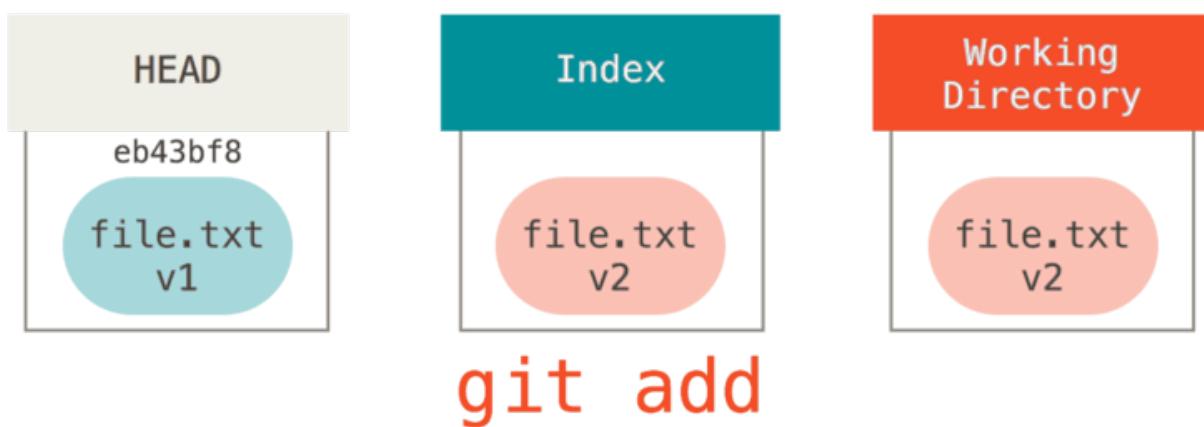
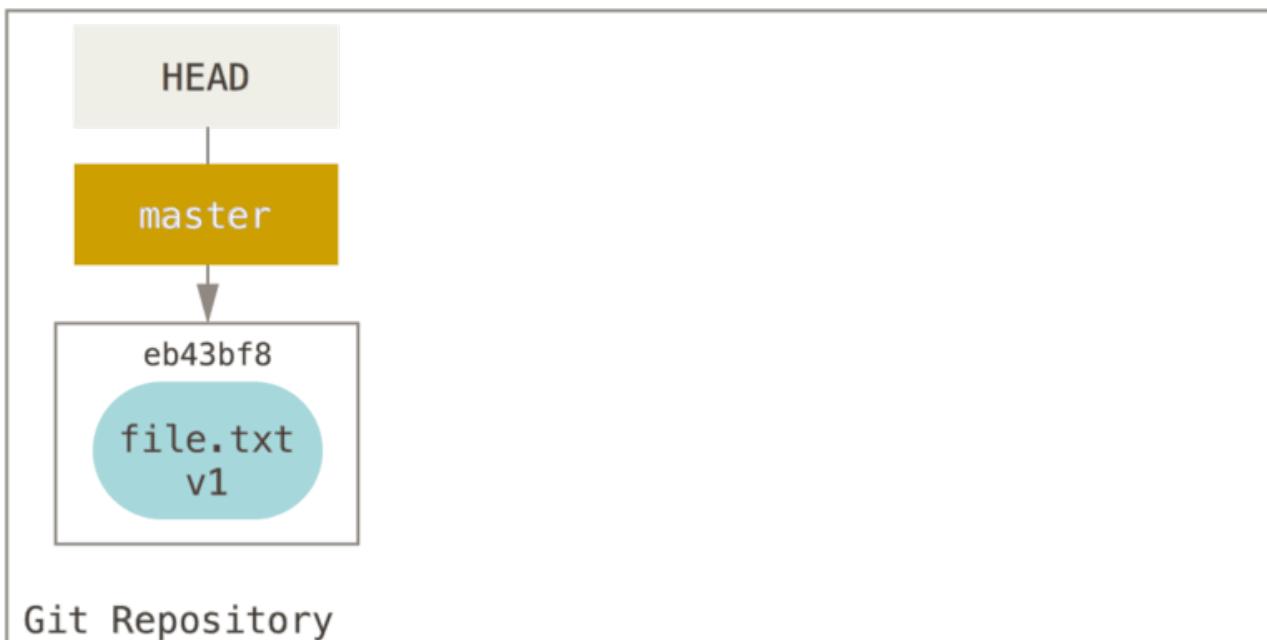
git commit

Si ejecutamos `git status`, no veremos ningún cambio, porque los tres árboles son iguales.

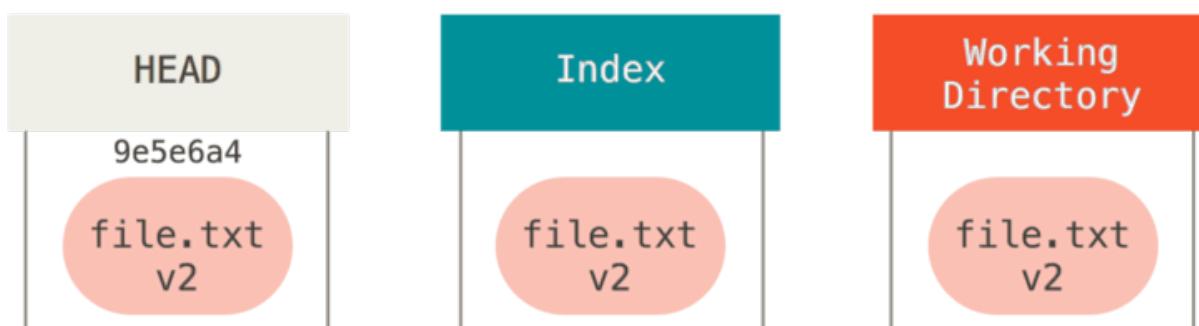
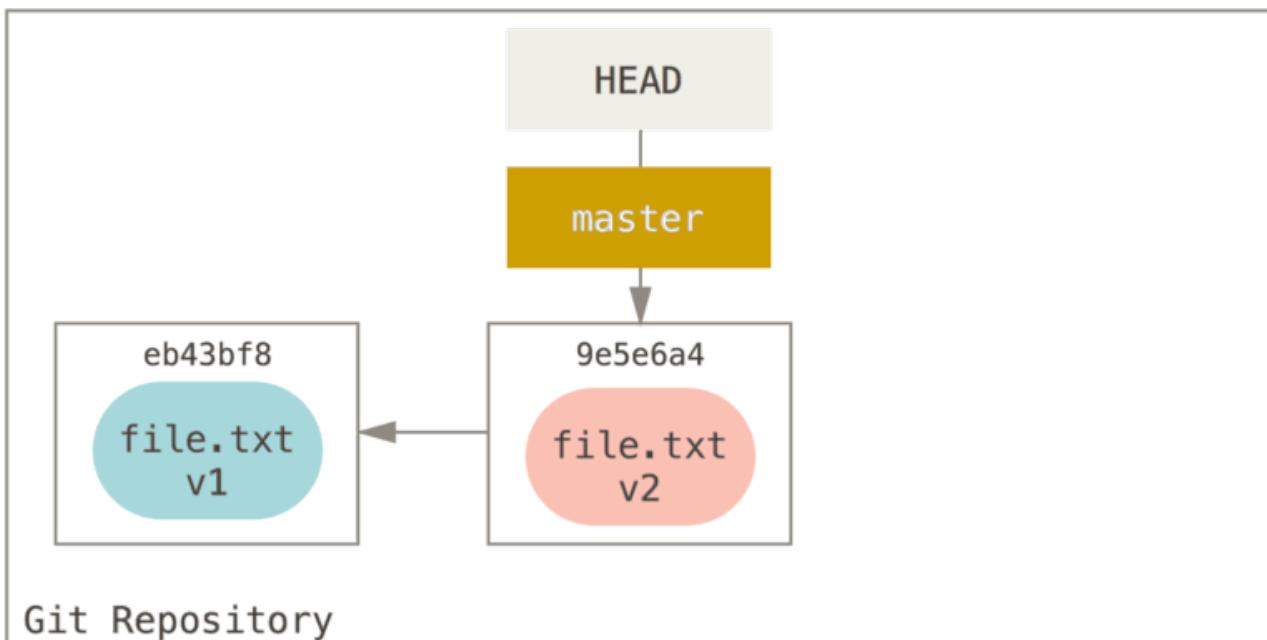
Ahora queremos hacer un cambio en ese archivo y hacerle un nuevo “commit”. Pasaremos por el mismo proceso; primero, cambiamos el archivo en nuestro directorio de trabajo. Llamemos a esto **v2** del archivo, y lo indicamos en rojo.



Si ejecutamos `git status` ahora, veremos el archivo en rojo como “Changes not staged for commit” porque esa entrada difiere entre el índice y el directorio de trabajo. A continuación, ejecutamos `git add` para ubicarlo en nuestro índice.



En este punto si ejecutamos `git status` veremos el archivo en verde debajo de “Changes to be committed” porque el Índice y el HEAD difieren – es decir, nuestro siguiente “commit” propuesto ahora es diferente de nuestro último “commit”. Finalmente, ejecutamos `git commit` para finalizar el “commit”.



git commit

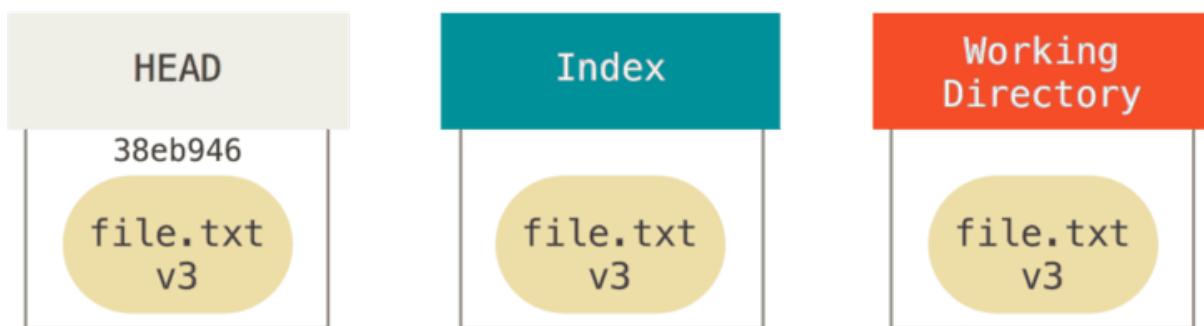
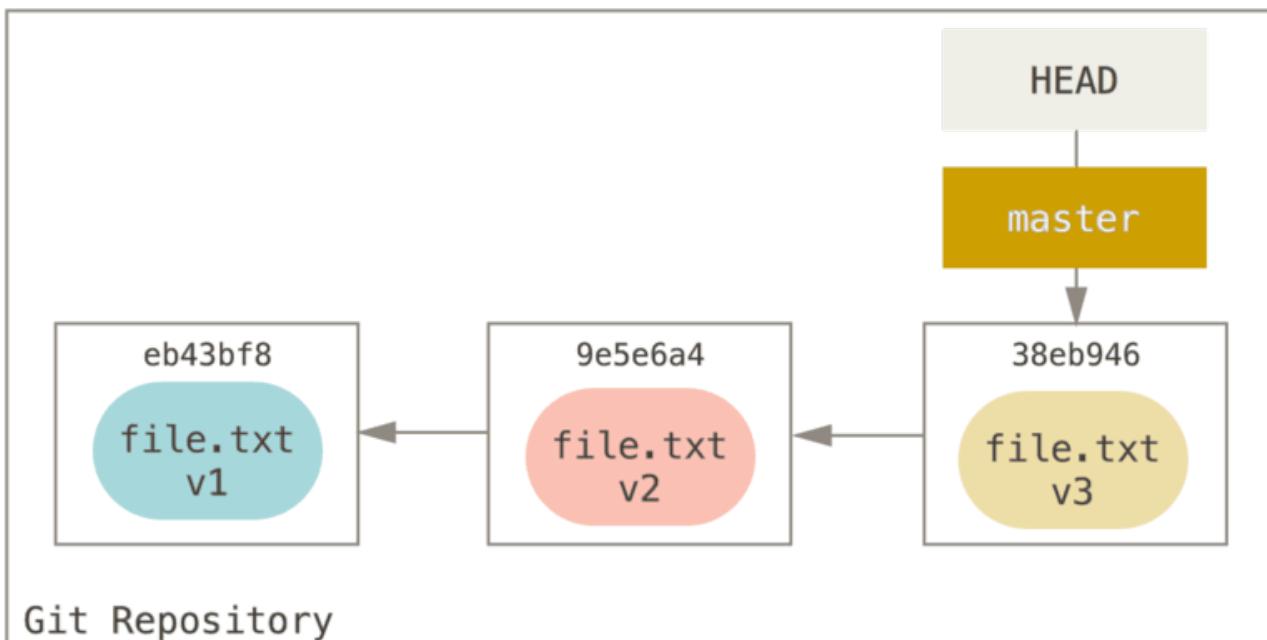
Ahora `git status` no nos dará salida, porque los tres árboles son iguales nuevamente.

El cambio de ramas o la clonación pasa por un proceso similar. Cuando verifica una rama, eso cambia `HEAD` para que apunte a la nueva “ref” de la rama, rellena su `Índice` con la instantánea de esa confirmación, luego copia los contenidos del `Índice` en tu `Directorio de Trabajo`.

El Papel del Reinicio

El comando `reset` tiene más sentido cuando se ve en este contexto.

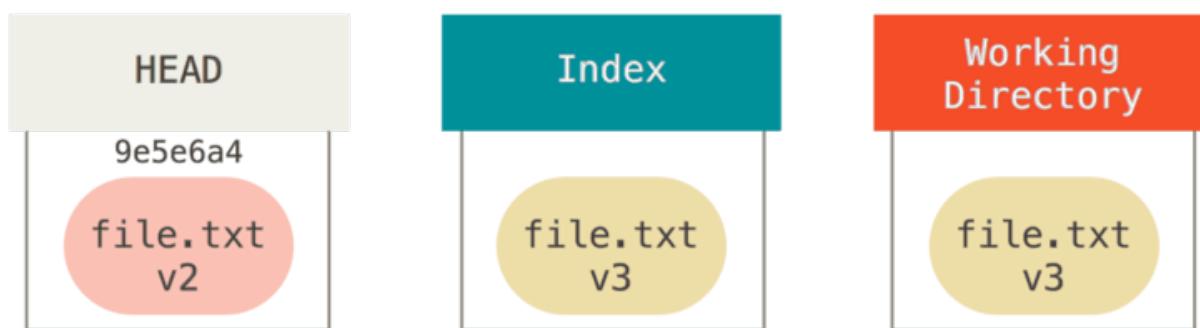
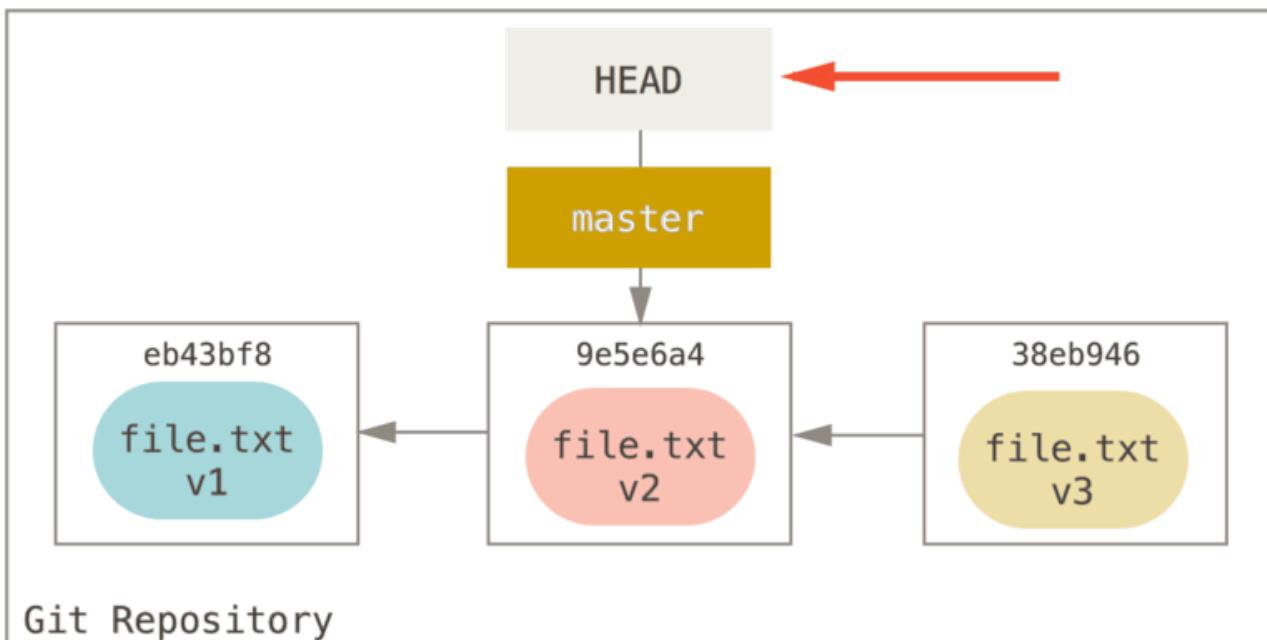
A los fines de estos ejemplos, digamos que hemos modificado `file.txt` de nuevo y le hemos hecho “commit” por tercera vez. Entonces ahora nuestra historia se ve así:



Hablemos ahora sobre lo que `reset` hace exactamente cuando es llamado. Manipula directamente estos tres árboles de una manera simple y predecible. Hace hasta tres operaciones básicas.

Paso 1: mover HEAD

Lo primero que `reset` hará es mover a lo que `HEAD` apunta. Esto no es lo mismo que cambiar `HEAD` en sí mismo (que es lo que hace `checkout`), `reset` mueve la rama a la que `HEAD` apunta. Esto significa que si `HEAD` está configurado en la rama `master` (es decir, estás actualmente en la rama `master`), ejecutar `git reset 9e5e64a` comenzará haciendo que `master` apunte a `9e5e64a`.



git reset --soft HEAD~

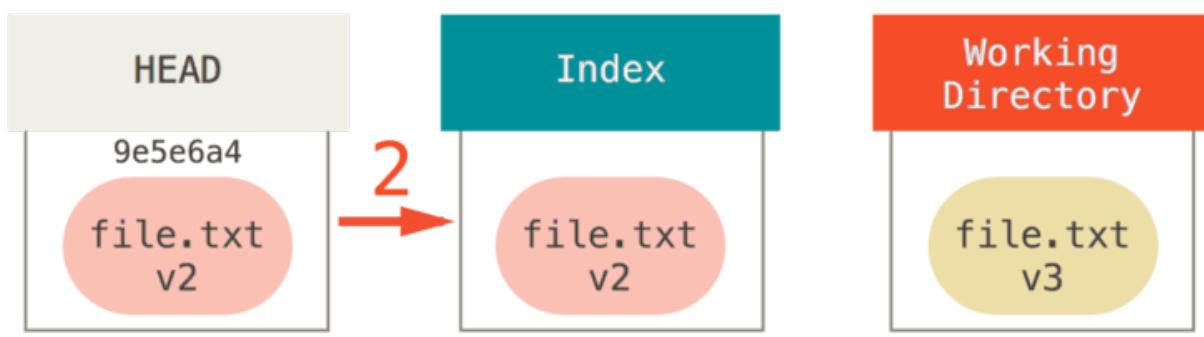
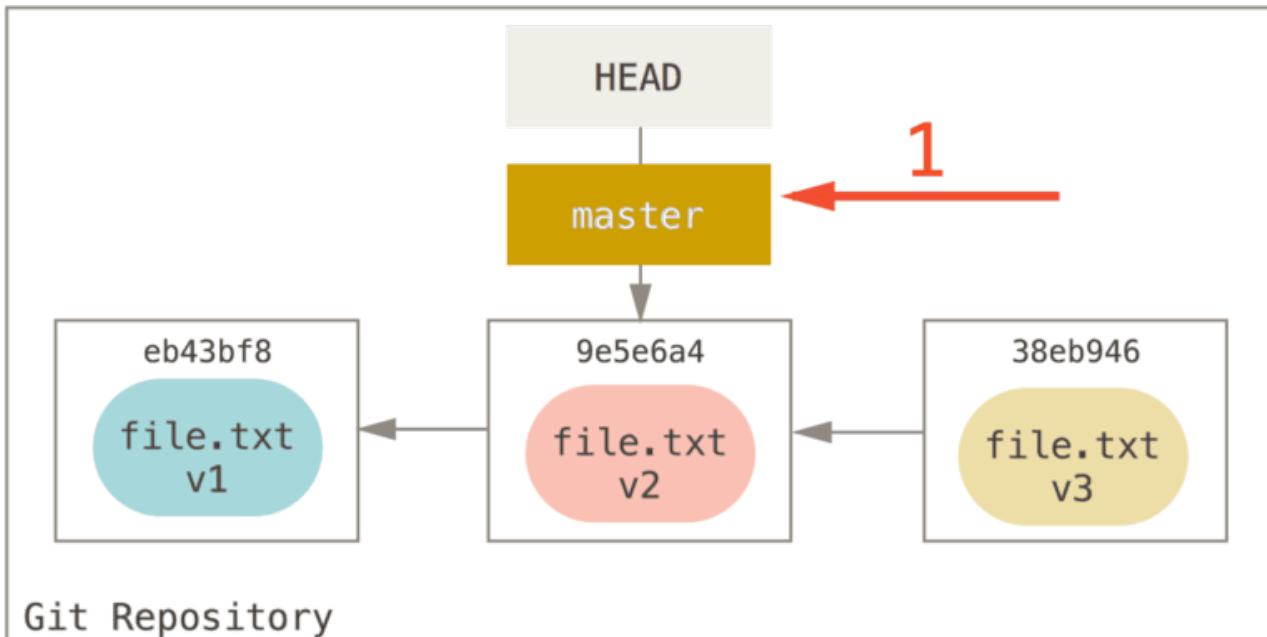
No importa qué forma de `reset` invoques con un ``commit, esto es lo primero que siempre intentará hacer. Con `reset --soft`, simplemente se detendrá allí.

Ahora tómate un segundo para mirar ese diagrama y darte cuenta de lo que sucedió: esencialmente deshizo el último comando `git commit`. Cuando ejecutas `git commit`, Git crea una nueva confirmación y mueve la rama a la que apunta `HEAD`. Cuando haces `reset` de vuelta a `HEAD~` (el padre de `HEAD`), está volviendo a colocar la rama donde estaba, sin cambiar el **Índice** o el Directorio de Trabajo. Ahora puedes actualizar el **Índice** y ejecutar `git commit` nuevamente para lograr lo que `git commit --amend` hubiera hecho (ver [Cambiando la última confirmación](#)).

Paso 2: Actualizando el índice (`--mixed`)

Ten en cuenta que si ejecutas `git status` ahora, verás en verde la diferencia entre el **Índice** y lo que el nuevo `HEAD` es.

Lo siguiente que `reset` hará es actualizar el **Índice** con los contenidos de cualquier instantánea que `HEAD` señale ahora.

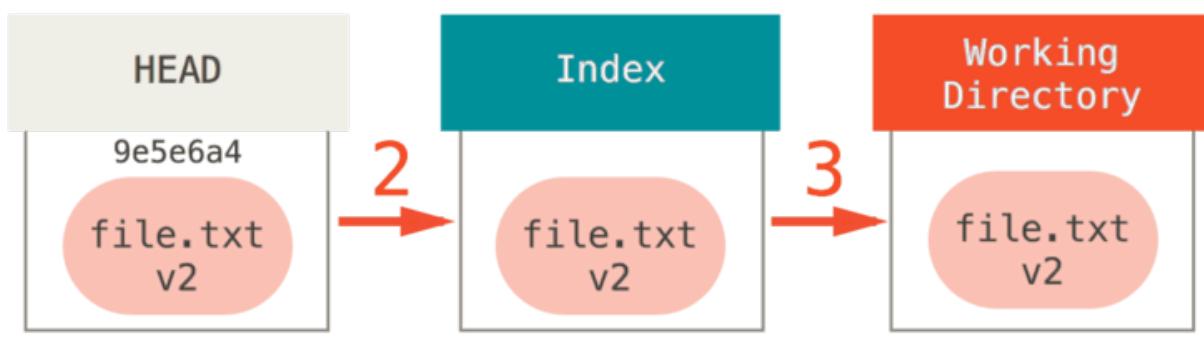
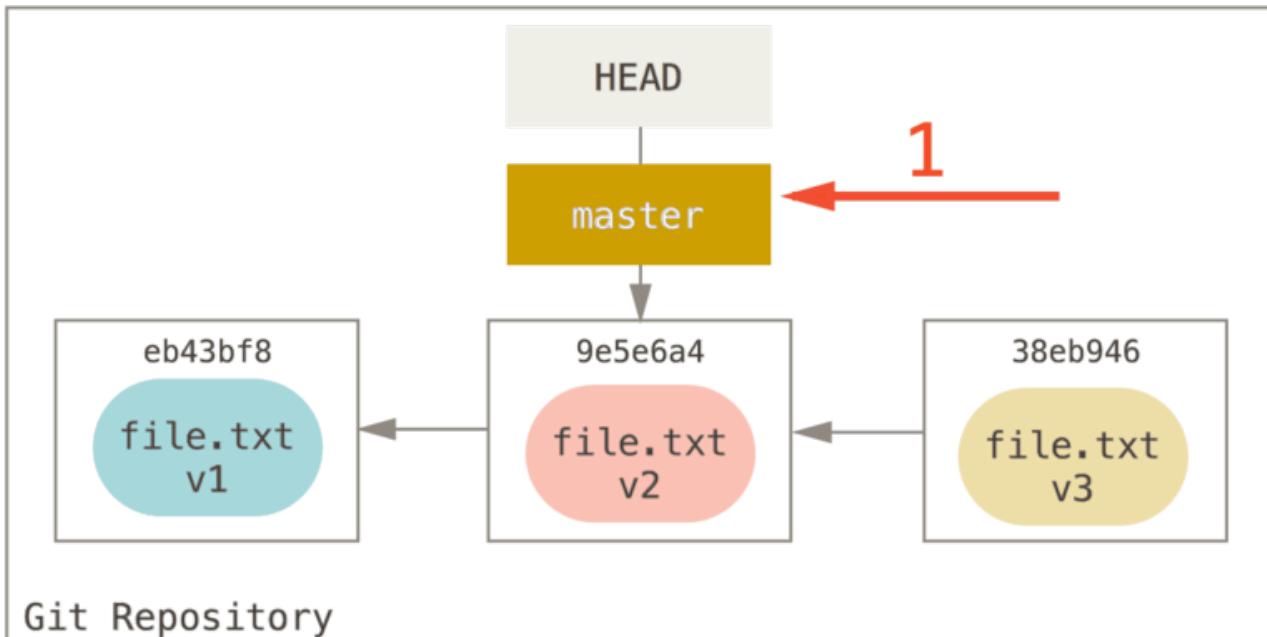


Si especificas la opción `--mixed`, `reset` se detendrá en este punto. Este también es el comportamiento por defecto, por lo que si no especificas ninguna opción (sólo `git reset HEAD~`, en este caso), aquí es donde el comando se detendrá.

Ahora tómate otro segundo para mirar ese diagrama y darte cuenta de lo que sucedió: deshizo tu último `commit` y también hizo `unstaged` de todo. Retrocedió a antes de ejecutar todos los comandos `git add` y `git commit`.

Paso 3: Actualizar el Directorio de Trabajo (--hard)

Lo tercero que `reset` hará es hacer que el **Directorio de Trabajo** se parezca al **Índice**. Si usas la opción `--hard`, continuará en esta etapa.



Entonces, pensemos en lo que acaba de pasar. Deshizo tu último commit, los comandos `git add` y `git commit`, y todo el trabajo que realizaste en tu **Directorio de Trabajo**.

Es importante tener en cuenta que este indicador (`--hard`) es la única manera de hacer que el comando `reset` sea peligroso, y uno de los pocos casos en que Git realmente destruirá los datos. Cualquier otra invocación de `reset` puede deshacerse fácilmente, pero la opción `--hard` no puede, ya que sobrescribe forzosamente los archivos en el **Directorio de Trabajo**. En este caso particular, todavía tenemos la versión **v3** de nuestro archivo en un “commit” en nuestro **DB** de Git, y podríamos recuperarla mirando nuestro **reflog**, pero si no le hubiéramos hecho “commit”, Git hubiese sobrescrito el archivo y sería irrecuperable.

Resumen

El comando `reset` sobrescribe estos tres árboles en un orden específico, deteniéndose cuando se le dice:

1. Mueva los puntos HEAD de la rama a (*deténgase aquí si --soft*)
2. Haga que el Índice se vea como HEAD (*deténgase aquí a menos que --hard*)

3. Haga que el Directorio de Trabajo se vea como el Índice

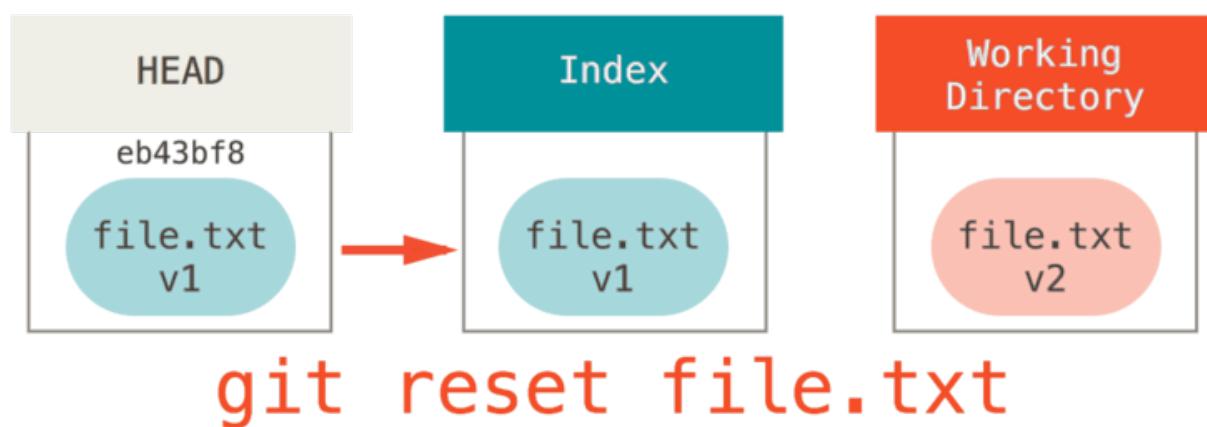
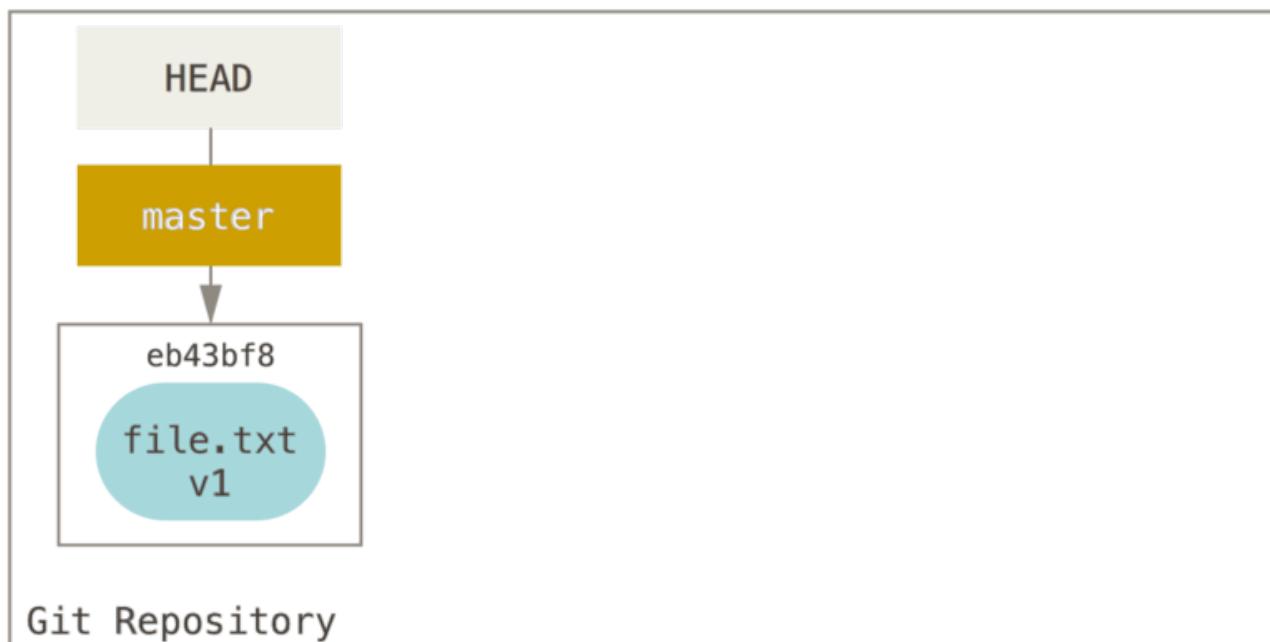
Reiniciar Con una Ruta

Eso cubre el comportamiento de `reset` en su forma básica, pero también puedes proporcionarle una ruta para actuar. Si especificas una ruta, `reset` omitirá el paso 1 y limitará el resto de sus acciones a un archivo o conjunto específico de archivos. Esto realmente tiene sentido – HEAD es solo un puntero, y no se puede apuntar a sólo una parte de un “commit” y otra parte de otro. Pero el **Índice** y el **Directorio de Trabajo** *pueden* actualizarse parcialmente, por lo que el reinicio continúa con los pasos 2 y 3.

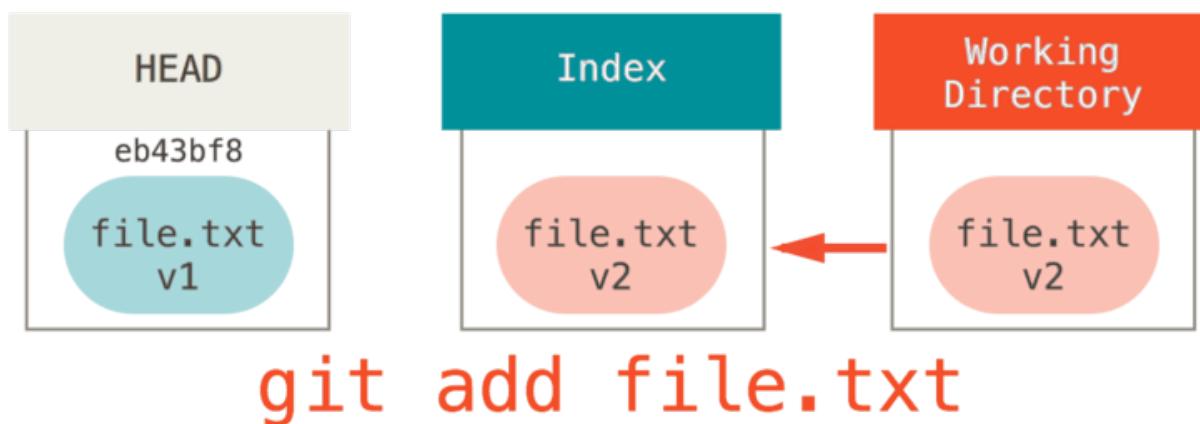
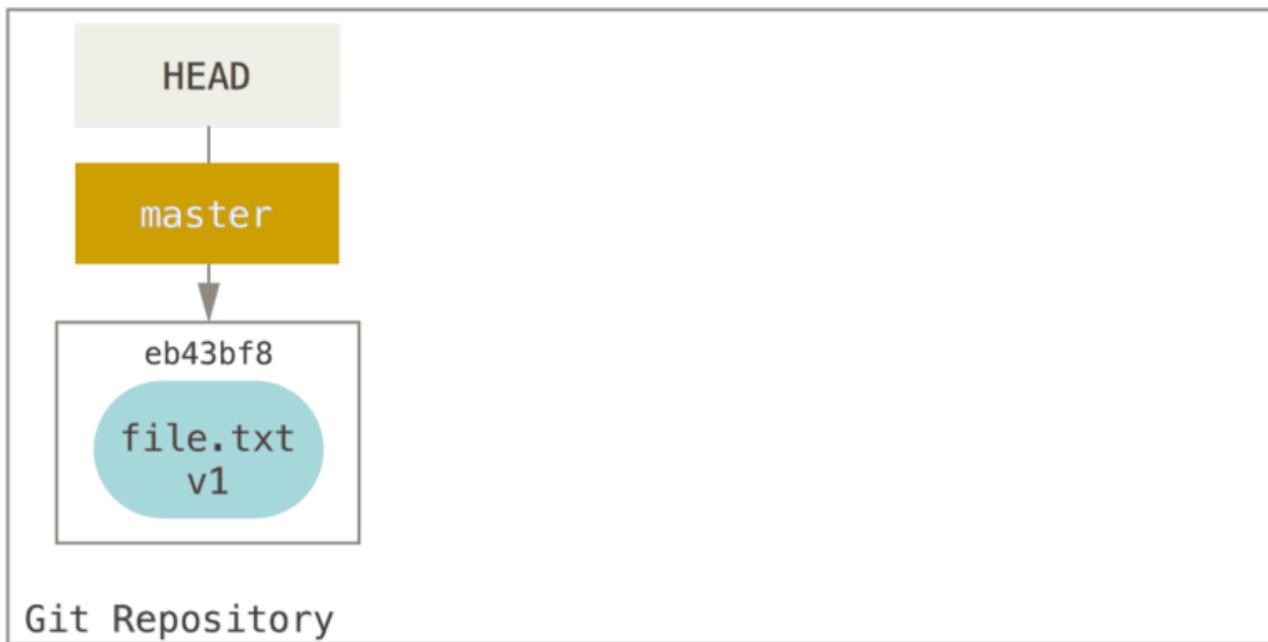
Entonces, supongamos que ejecutamos `git reset file.txt`. Este formulario (ya que no especificó un commit SHA-1 o una rama, y no especificó `--soft` o `--hard`) es una abreviatura de `git reset --mixed HEAD file.txt`, la cual hará:

1. Mueva los puntos HEAD de la rama a (*omitido*)
2. Haga que el Índice se vea como HEAD (*deténgase aquí*)

Por lo tanto, básicamente solo copia `archivo.txt` de **HEAD** al **Índice**.

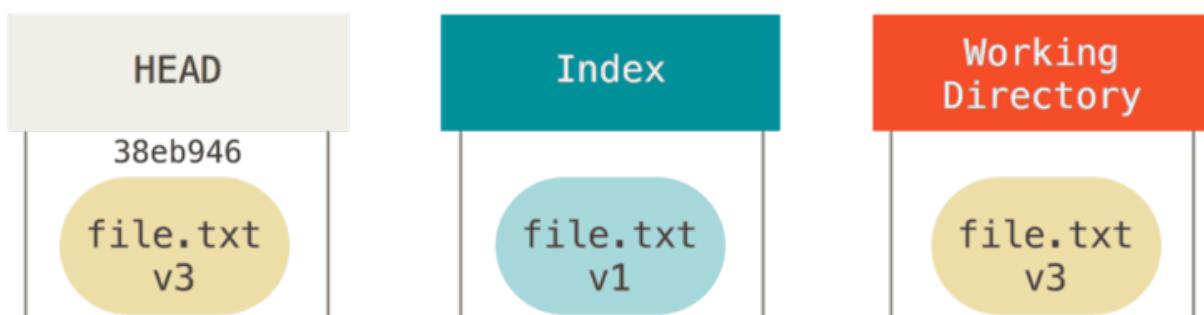
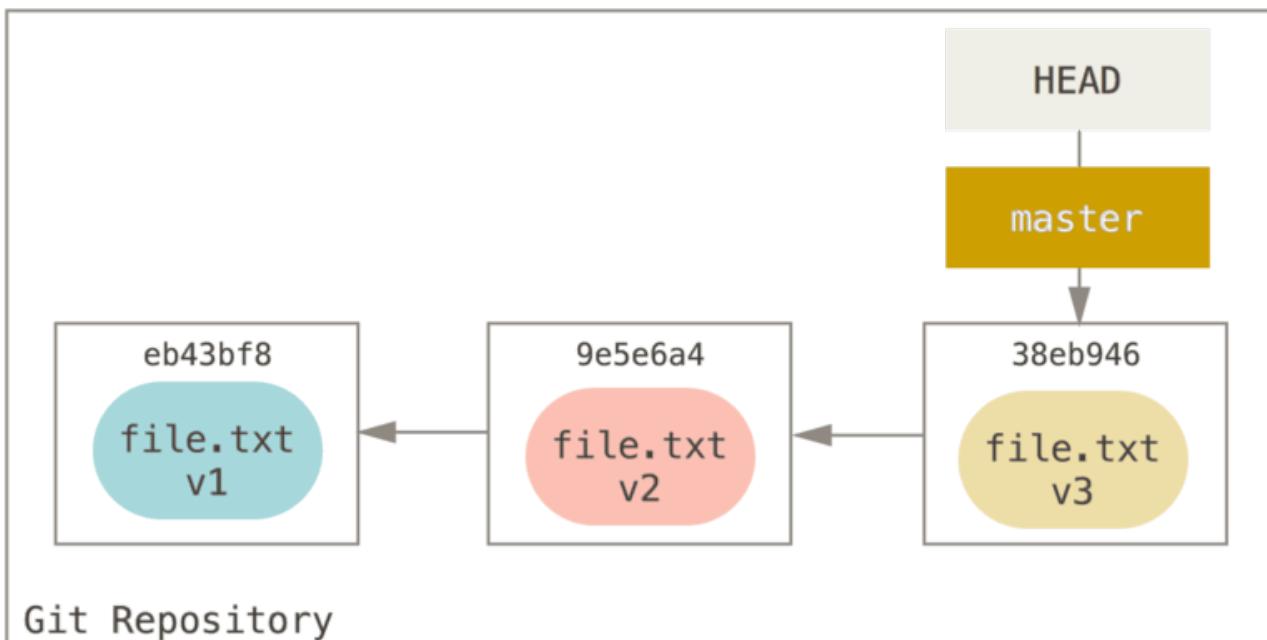


Esto tiene el efecto práctico de hacer *unstaging* al archivo. Si miramos el diagrama para ese comando y pensamos en lo que hace `git add`, son exactamente opuestos.



Esta es la razón por la cual el resultado del comando `git status` sugiere que ejecute esto para descentralizar un archivo. (Consulte [Deshacer un Archivo Preparado](#) para más sobre esto).

Igualmente podríamos no permitir que Git suponga que queríamos “extraer los datos de HEAD” especificando un “commit” específico para extraer esa versión del archivo. Simplemente ejecutaríamos algo como `git reset eb43bf file.txt`.



git reset eb43 -- file.txt

Esto efectivamente hace lo mismo que si hubiéramos revertido el contenido del archivo a **v1** en el **Directorio de Trabajo**, ejecutado `git add` en él, y luego lo revertimos a **v3** nuevamente (sin tener que ir a través de todos esos pasos). Si ejecutamos `git commit` ahora, registrará un cambio que revierte ese archivo de vuelta a **v1**, aunque nunca más lo volvimos a tener en nuestro **Directorio de Trabajo**.

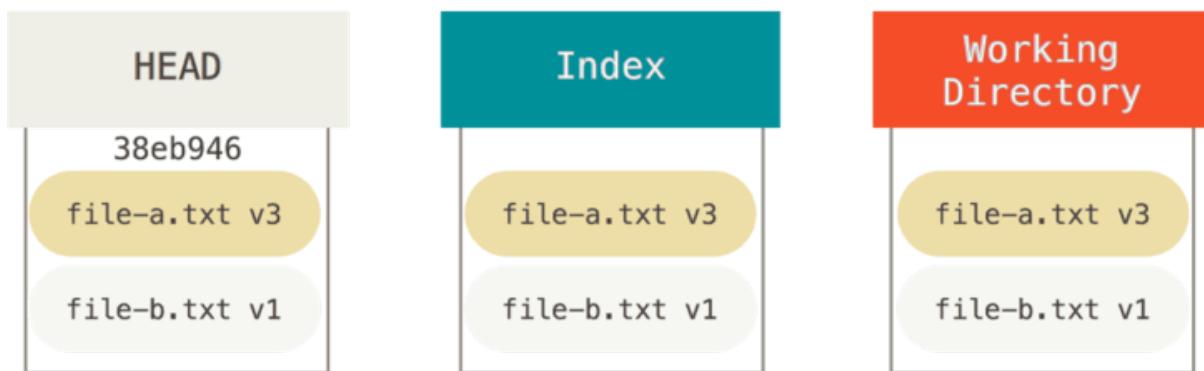
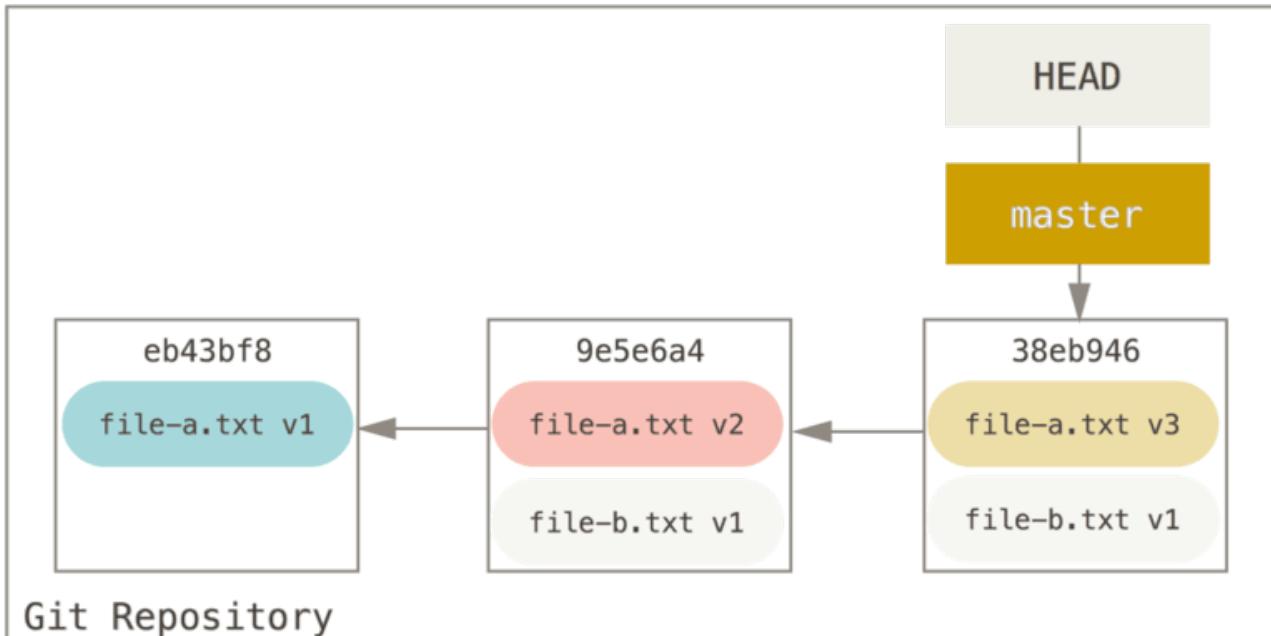
También es interesante observar que, como `git add`, el comando `reset` aceptará una opción `--patch` para hacer *unstage* del contenido en una base hunk-by-hunk. Por lo tanto, puede hacer *unstage* o revertir el contenido de forma selectiva.

Aplastando

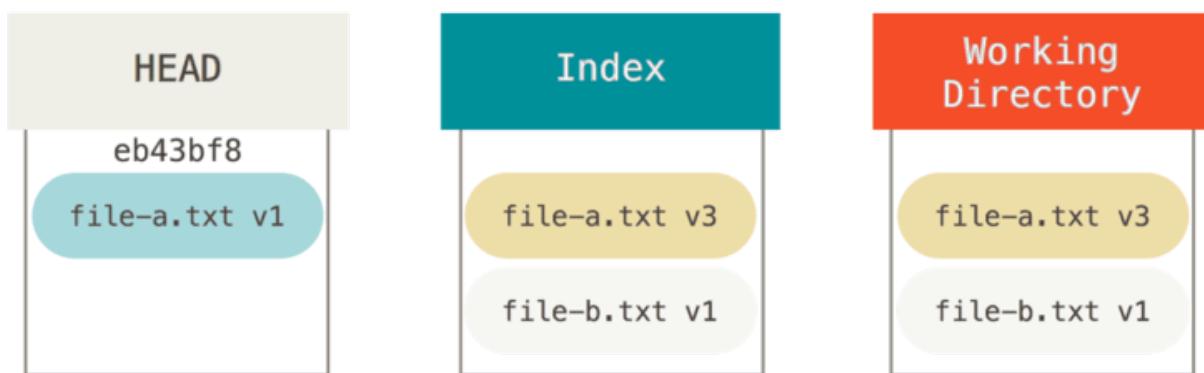
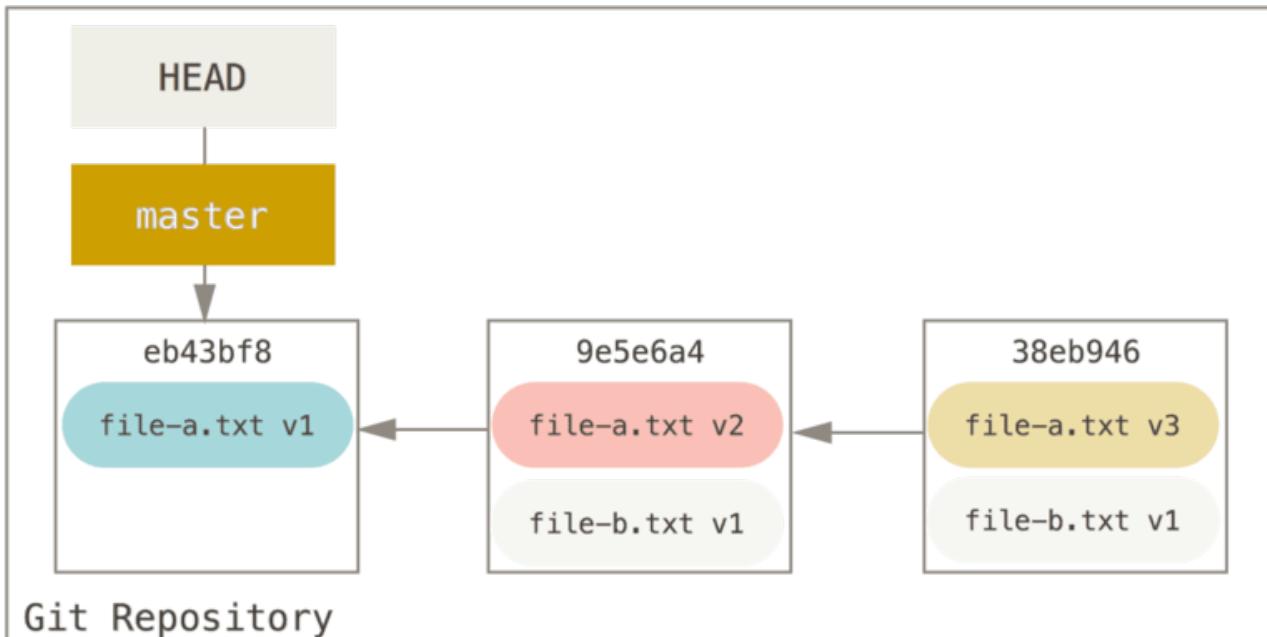
Veamos cómo hacer algo interesante con este poder recién descubierto – aplastando “commits”.

Supongamos que tienes una serie de confirmaciones con mensajes como “oops.”, “WIP” y “se olvidó de este archivo”. Puedes usar `reset` para aplastarlos rápida y fácilmente en una sola confirmación que lo hace ver realmente inteligente. ([Aplastando](#) muestra otra forma de hacerlo, pero en este ejemplo es más simple usar `reset`.)

Supongamos que tiene un proyecto en el que el primer “commit” tiene un archivo, el segundo “commit” agregó un nuevo archivo y cambió el primero, y el tercer “commit” cambió el primer archivo otra vez. El segundo “commit” fue un trabajo en progreso y quieres aplastarlo.

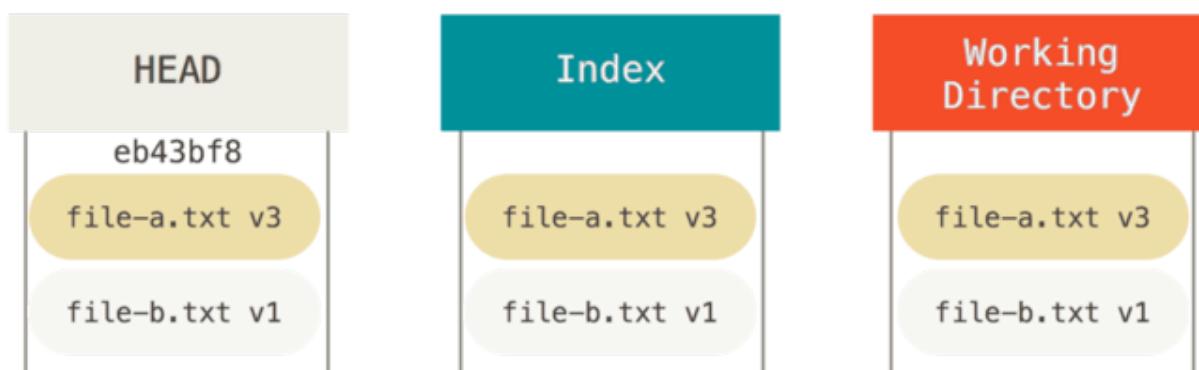
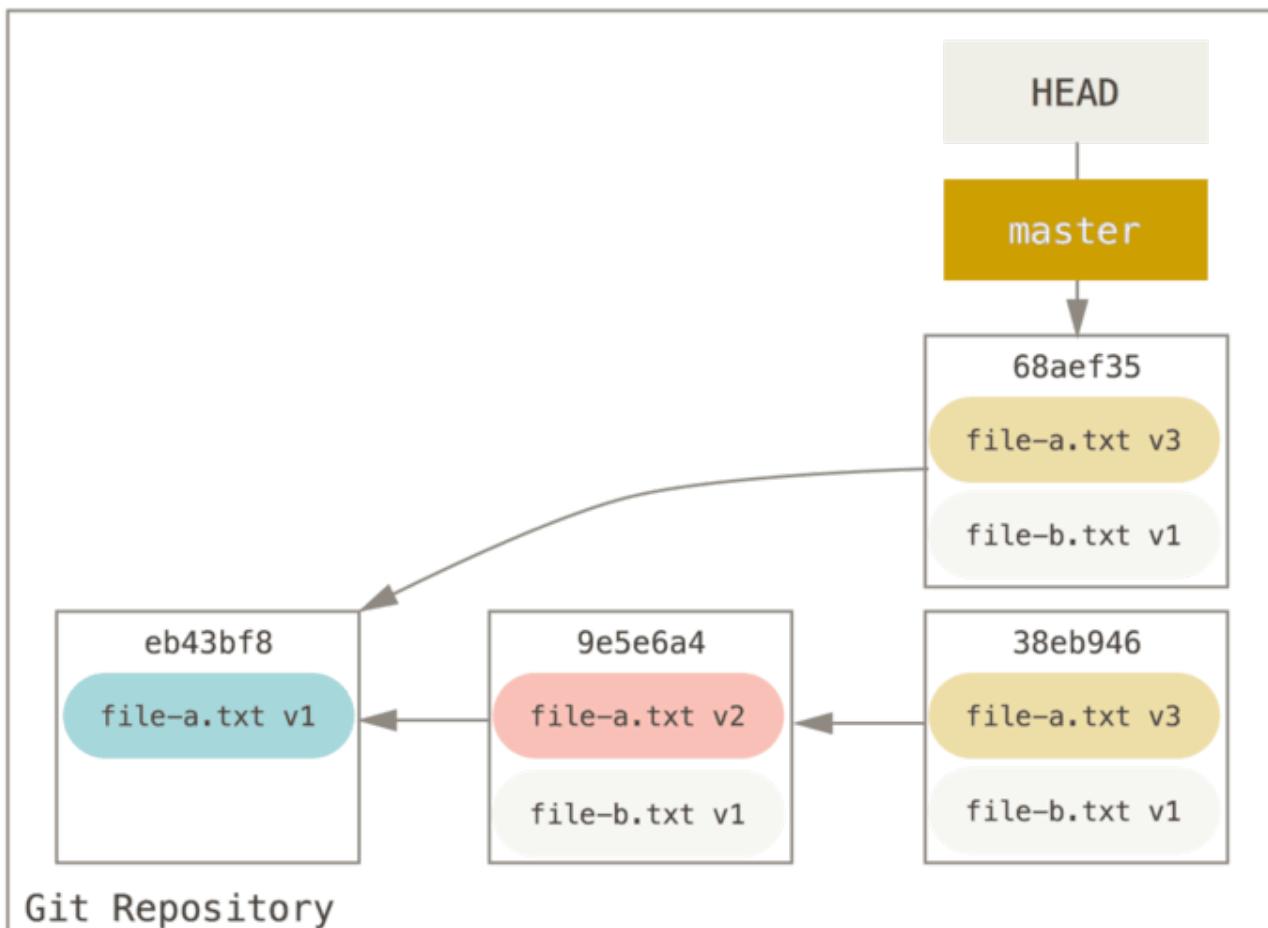


Puedes ejecutar `git reset --soft HEAD~2` para mover la rama HEAD a un “commit” anterior (el primer “commit” que deseas mantener):



git reset --soft HEAD~2

Y luego simplemente ejecuta `git commit` nuevamente:



git commit

Ahora puedes ver que el historial alcanzable, la historia que empujarías, ahora parece que tuvo un “commit” con `archivo-a.txt` v1, luego un segundo que ambos modificaron `archivo-a.txt` a v3 y agregaron `archivo-b.txt`. El “commit” con la versión v2 del archivo ya no está en el historial.

Echale Un vistazo

Finalmente, puedes preguntarte cuál es la diferencia entre `checkout` y `reset`. Al igual que `reset`, `checkout` manipula los tres árboles, y es un poco diferente dependiendo de si se le da al comando una ruta de archivo o no.

Sin Rutas

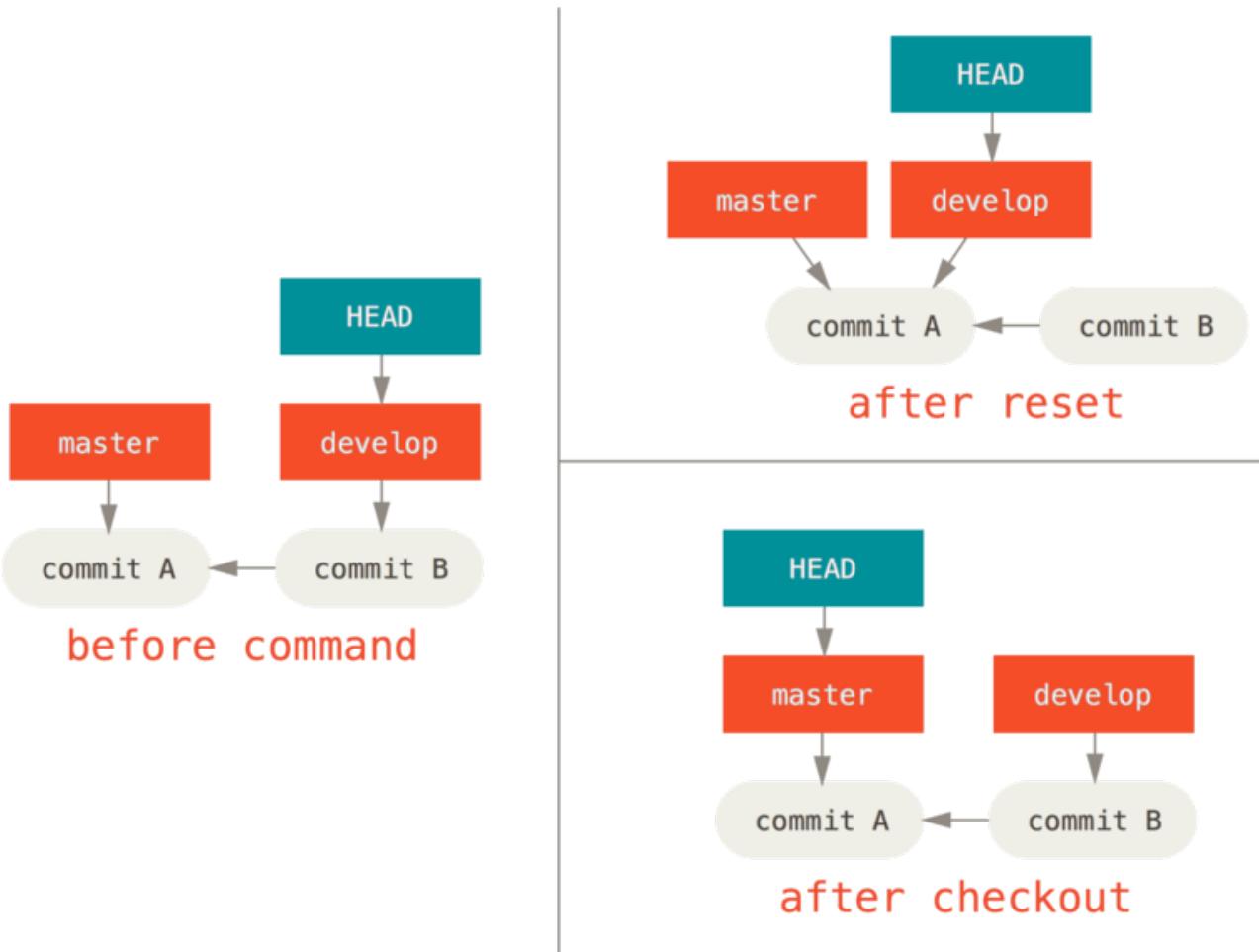
Ejecutar `git checkout [branch]` es bastante similar a ejecutar `git reset --hard [branch]` porque actualiza los tres árboles para que se vea como `[branch]`, pero hay dos diferencias importantes.

Primero, a diferencia de `reset --hard`, `checkout` está en el **directorio-de-trabajo** seguro; Verificará para asegurarse de que no está volando los archivos que tienen cambios en ellos. En realidad, es un poco más inteligente que eso – intenta hacer una fusión trivial en el **Directorio de Trabajo**, por lo que todos los archivos que *no hayan* cambiado serán actualizados. `reset --hard`, por otro lado, simplemente reemplazará todo en general sin verificar.

La segunda diferencia importante es cómo actualiza **HEAD**. Donde `reset` moverá la rama a la que **HEAD** apunta, `checkout` moverá **HEAD** para señalar otra rama.

Por ejemplo, digamos que tenemos las ramas `master` y `develop` que apuntan a diferentes *commits*, y actualmente estamos en `develop` (así que **HEAD** la señala). Si ejecutamos `git reset master`, `develop` ahora apuntará al mismo “commit” que `master`. Si en cambio ejecutamos `git checkout master`, `develop` no se mueve, **HEAD** sí lo hace. **HEAD** ahora apuntará a `master`.

Entonces, en ambos casos estamos moviendo **HEAD** para apuntar al “commit” A, pero el *cómo* lo hacemos es muy diferente. `reset` moverá los puntos **HEAD** de la rama A, `checkout` mueve el mismo **HEAD**.



Con Rutas

La otra forma de ejecutar `checkout` es con una ruta de archivo, que como `reset`, no mueve **HEAD**. Es como `git reset [branch] file` en que actualiza el índice con ese archivo en ese “commit”, pero también sobrescribe el archivo en el **Directorio de Trabajo**. Sería exactamente como `git reset --hard [branch] file` (si `reset` permitiera ejecutar eso) - no está directorio-de-trabajo seguro, y no mueve a **HEAD**.

Además, al igual que `git reset` y `git add`, `checkout` aceptará una opción `--patch` para permitir revertir selectivamente el contenido del archivo sobre una base hunk-by-hunk.

Resumen

Esperamos que ahora entiendas y te sientas más cómodo con el comando `reset`, pero probablemente todavía estés un poco confundido acerca de cómo exactamente difiere de `checkout` y posiblemente no puedas recordar todas las reglas de las diferentes invocaciones.

Aquí hay una hoja de trucos para cuáles comandos afectan a cuáles árboles. La columna “HEAD” dice “REF” si ese comando mueve la referencia (rama) a la que **HEAD** apunta, y “HEAD” si se mueve al propio **HEAD**. Presta especial atención a la columna **WD Safe**: si dice **NO**, tómate un segundo para pensar antes de ejecutar ese comando.

| | HEAD | Index | Workdir | WD Safe? |
|---------------------------------------|-------------|--------------|----------------|-----------------|
| Nivel de Commit | | | | |
| <code>reset --soft [commit]</code> | REF | NO | NO | SI |
| <code>reset [commit]</code> | REF | SI | NO | SI |
| <code>reset --hard [commit]</code> | REF | SI | SI | NO |
| <code>checkout [commit]</code> | HEAD | SI | SI | SI |
| Nivel de Archivo | | | | |
| <code>reset (commit) [file]</code> | NO | SI | NO | SI |
| <code>checkout (commit) [file]</code> | NO | SI | SI | NO |

Fusión Avanzada

La fusión en Git suele ser bastante fácil. Dado que Git facilita la fusión de otra rama varias veces, significa que puede tener una rama de larga duración, pero puede mantenerla actualizada sobre la marcha, resolviendo pequeños conflictos a menudo, en lugar de sorprenderse por un conflicto enorme en el final de la serie.

Sin embargo, a veces ocurren conflictos engañosos. A diferencia de otros sistemas de control de versiones, Git no intenta ser demasiado listo para fusionar la resolución de conflictos. La filosofía de Git es ser inteligente para determinar cuándo una resolución de fusión no es ambigua, pero si hay un conflicto, no intenta ser inteligente para resolverlo automáticamente. Por lo tanto, si espera demasiado para fusionar dos ramas que divergen rápidamente, puede encontrarse con algunos problemas.

En esta sección, veremos cuáles podrían ser algunos de esos problemas y qué herramientas le dará Git para ayudarlo a manejar estas situaciones más engañosas. También cubriremos algunos de los diferentes tipos de fusión no estándar que puede hacer, y también veremos cómo deshacerse de las fusiones que ha realizado.

Conflictos de Fusión

Si bien cubrimos algunos conceptos básicos para resolver conflictos de fusión en [Principales Conflictos que Pueden Surgir en las Fusiones](#), para conflictos más complejos, Git proporciona algunas herramientas para ayudarlo a descubrir qué está sucediendo y cómo lidiar mejor con el conflicto.

En primer lugar, si es posible, intente asegurarse de que su directorio de trabajo esté limpio antes de realizar una fusión que pueda tener conflictos. Si tiene un trabajo en progreso, hágale commit a una rama temporal o stash. Esto hace que pueda deshacer **cualquier cosa** que intente aquí. Si tiene cambios no guardados en su directorio de trabajo cuando intenta fusionarlos, algunos de estos consejos pueden ayudarlo a perder ese trabajo.

Veamos un ejemplo muy simple. Tenemos un archivo Ruby super simple que imprime *hello world*.

```
#!/usr/bin/env ruby

def hello
    puts 'hello world'
end

hello()
```

En nuestro repositorio, creamos una nueva rama llamada `whitespace` y procedemos a cambiar todas las terminaciones de línea de Unix a terminaciones de línea de DOS, esencialmente cambiando cada línea del archivo, pero solo con espacios en blanco. Luego cambiamos la línea "hello world" a "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(-)
```

Ahora volvemos a nuestra rama `master` y agregamos cierta documentación para la función.

```
$ 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(+)
```

Ahora tratamos de fusionarnos en nuestra rama `whitespace` y tendremos conflictos debido a los cambios en el espacio en blanco.

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

Abortar una Fusión

Ahora tenemos algunas opciones. Primero, cubramos cómo salir de esta situación. Si tal vez no esperabas conflictos y aún no quieres lidiar con la situación, simplemente puedes salir de la fusión con `git merge --abort`.

```
$ git status -sb
## master
UU hello.rb

$ git merge --abort

$ git status -sb
## master
```

La opción `git merge --abort` intenta volver a su estado antes de ejecutar la fusión. Los únicos casos en los que podría no ser capaz de hacer esto a la perfección serían si hubiera realizado cambios sin stash, no confirmados en su directorio de trabajo cuando

lo ejecutó, de lo contrario, debería funcionar bien.

Si por alguna razón se encuentra en un estado horrible y solo quiere comenzar de nuevo, también puede ejecutar `git reset --hard HEAD` o donde quiera volver. Recuerde, una vez más, que esto hará volar su directorio de trabajo, así que asegúrese de no querer ningún cambio allí.

Ignorando el Espacio en Blanco

En este caso específico, los conflictos están relacionados con el espacio en blanco. Sabemos esto porque el caso es simple, pero también es muy fácil saberlo en casos reales, al analizar el conflicto, porque cada línea se elimina por un lado y se agrega nuevamente por el otro. De manera predeterminada, Git ve que todas estas líneas están siendo modificadas, por lo que no puede fusionar los archivos.

Sin embargo, la estrategia de combinación predeterminada puede tomar argumentos, y algunos de ellos son acerca de ignorar adecuadamente los cambios del espacio en blanco. Si ve que tiene muchos problemas con espacios en blanco en una combinación, simplemente puede cancelarla y volverla a hacer, esta vez con `-Xignore-all-space` o ``-Xignore-space-change``. La primera opción ignora los cambios en cualquier **cantidad** de espacios en blanco existentes, la segunda ignora por completo todos los cambios de espacios en blanco.

```
$ 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(-)
```

Dado que en este caso, los cambios reales del archivo no eran conflictivos, una vez que ignoramos los cambios en los espacios en blanco, todo se fusiona perfectamente.

Esto es un salvavidas si tiene a alguien en su equipo a quien le gusta ocasionalmente reformatear todo, desde espacios hasta pestañas o viceversa.

Re-fusión Manual de Archivos

Aunque Git maneja muy bien el preprocessamiento de espacios en blanco, hay otros tipos de cambios que quizás Git no pueda manejar de manera automática, pero que son correcciones de secuencias de comandos. Como ejemplo, imaginemos que Git no pudo manejar el cambio en el espacio en blanco y que teníamos que hacerlo a mano.

Lo que realmente tenemos que hacer es ejecutar el archivo que intentamos fusionar a través de un programa `dos2unix` antes de intentar fusionar el archivo. Entonces, ¿cómo haríamos eso?

Primero, entramos en el estado de conflicto de la fusión. Luego queremos obtener copias de mi versión del archivo, su versión (de la rama en la que nos estamos fusionando) y la versión común (desde donde ambos lados se bifurcaron). Entonces, queremos arreglar

su lado o nuestro lado y volver a intentar la fusión sólo para este único archivo.

Obtener las tres versiones del archivo es bastante fácil. Git almacena todas estas versiones en el índice bajo “etapas”, cada una de las cuales tiene números asociados. La etapa 1 es el ancestro común, la etapa 2 es su versión y la etapa 3 es de la `MERGE_HEAD`, la versión en la que se está fusionando (“suya”).

Puede extraer una copia de cada una de estas versiones del archivo en conflicto con el comando `git show` y una sintaxis especial.

```
$ git show :1:hello.rb > hello.common.rb  
$ git show :2:hello.rb > hello.ours.rb  
$ git show :3:hello.rb > hello.theirs.rb
```

Si quiere ponerse un poco más intenso, también puede usar el comando de plomería `ls-files -u` para obtener el verdadero SHA-1s de las manchas de Git para cada uno de los archivos.

```
$ git ls-files -u  
100755 ac51efdc3df4f4fd328d1a02ad05331d8e2c9111 1 hello.rb  
100755 36c06c8752c78d2aff89571132f3bf7841a7b5c3 2 hello.rb  
100755 e85207e04dfdd5eb0a1e9febbc67fd837c44a1cd 3 hello.rb
```

El `:1:hello.rb` es solo una clave para buscar esa mancha SHA-1.

Ahora que tenemos el contexto de estas tres etapas en nuestro directorio de trabajo, manualmente podemos arreglarlos para solucionar los problemas de espacios en blanco y volver a fusionar el archivo con el poco conocido comando `git merge-file` que hace exactamente eso.

```

$ 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()

```

En este punto hemos, agradablemente, fusionado el archivo. De hecho, esto en realidad funciona mejor que la opción de `ignore-all-space`, porque realmente soluciona los cambios de los espacios en blanco antes de la fusión, en lugar de simplemente ignorarlo. En la fusión `ignore-all-space`, en realidad, terminamos con unas pocas líneas con finales de línea DOS, haciendo que las cosas se mezclen.

Si quiere tener una idea antes de finalizar este compromiso sobre qué había cambiado en realidad entre un lado y el otro, puede pedirle a `git diff` que compare qué hay en su directorio de trabajo que está a punto de comprometer como resultado de la fusión a cualquiera de estas etapas. Vamos a través de todas ellas.

Para comparar el resultado con lo que tenías en su rama antes de la fusión, en otras palabras, para ver lo que su fusión insertó, puede correr `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()
```

Así, podemos observar fácilmente lo que sucedió en nuestra rama, y si lo que en realidad estamos insertando a este archivo con esta fusión está cambiando solamente esa línea.

Si queremos ver cómo el resultado de la fusión difiere de lo que estaba del otro lado, podemos correr `git diff --theirs`. En este y el siguiente ejemplo, tenemos que usar `-w` para despojarlo de los espacios en blanco porque lo estamos comparando con lo que está en Git, no con nuestro archivo limpio `hello.theirs.rb`

```
$ 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
```

Finalmente, puede observar cómo el archivo ha cambiado desde ambos lados con `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()

```

En este punto podemos usar el comando `git clean` para limpiar los archivos sobrantes que creamos para hacer la fusión manual, pero que ya no necesitamos.

```

$ git clean -f
Removing hello.common.rb
Removing hello.ours.rb
Removing hello.theirs.rb

```

Revisando Los Conflictos

Tal vez en este punto no estemos felices con la resolución por alguna razón, o quizás manualmente editando uno o ambos lados todavía no funciona como es debido y necesitamos más contexto.

Cambiemos el ejemplo un poco. En este caso, tenemos dos ramas de larga vida las cuales cada una tiene unos pocos “commit” en ella, aparte de crear un contenido de conflicto legítimo cuando es fusionado.

```

$ 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

```

Ahora tenemos tres “commit” únicos que viven solo en la rama `principal` y otros tres que viven en la rama `mundo`. Si intentamos fusionar la rama `mundo`, generaremos un

conflicto.

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

Nos gustaría ver cuál es el conflicto de fusión. Si abrimos el archivo, veremos algo así:

```
#! /usr/bin/env ruby

def hello
<<<<< HEAD
  puts 'hola world'
=====
  puts 'hello mundo'
>>>>> mundo
end

hello()
```

Ambos lados de la fusión han añadido contenido a este archivo, pero algunos de los “commit” han modificado el archivo en el mismo lugar que causó el conflicto.

Exploraremos un par de herramientas que ahora tiene a su disposición para determinar cómo el conflicto resultó ser. Tal vez, no es tan obvio cómo exactamente debería solucionar este problema. Necesita más contexto.

Una herramienta útil es `git checkout` con la opción “`--conflict`”. Esto revisará el archivo de nuevo y reemplazará los marcadores de conflicto de la fusión. Esto puede ser útil si quiere reiniciar los marcadores y tratar de resolverlos de nuevo.

Puedes pasar `--conflict` en lugar de `diff3` o `merge` (lo que es por defecto). Si pasa `diff3`, Git usará una versión un poco diferente de marcadores de conflicto, no solo dándole “ours” versión y la versión de “theirs”, sino también la versión “base” en línea para darle más contexto.

```
$ git checkout --conflict=diff3 hello.rb
```

Una vez que corremos eso, en su lugar el archivo se verá así:

```

#!/usr/bin/env ruby

def hello
<<<<< ours
  puts 'hola world'
||||||| base
  puts 'hello world'
=====
  puts 'hello mundo'
>>>>> theirs
end

hello()

```

Si este formato es de su agrado, puede configurarlo como “default” para futuros conflictos de fusión al colocar el `merge.conflictstyle` configurándolo a `diff3`.

```
$ git config --global merge.conflictstyle diff3
```

El comando `git checkout` puede también tomar la opción de `--theirs`o la `--ours`, lo cual puede ser una manera mucho más rápida de escoger un lado o el otro sin tener que fusionar las cosas en lo absoluto.

Esto puede ser particularmente útil para conflictos de archivos binarios donde simplemente puede escoger un lado, o donde solo quiere fusionar ciertos archivos desde otra rama – puede hacer la fusión y luego revisar ciertos archivos de un lado o del otro antes de comprometerlos

Registro de Fusión

Otra herramienta útil al resolver conflictos de fusión es `git log`. Esto puede ayudarle a tener contexto de lo que pudo haber contribuido a los conflictos. Revisar un poco el historial para recordar por qué dos líneas de desarrollo estaban tocando el mismo código de área, puede ser muy útil algunas veces.

Para obtener una lista completa de “commit” únicos que fueron incluidos en cualquiera de las ramas involucradas en esta fusión, podemos usar la sintaxis “triple dot” (triple punto) que aprendimos en [Tres puntos](#).

```

$ 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

```

Esa es una buena lista de los seis compromisos involucrados, así como en qué línea de desarrollo estuvo cada compromiso.

Sin embargo, podemos simplificar aún más esto para darnos un contexto mucho más específico. Si añadimos la opción `--merge` a `git log`, solo mostrará los compromisos en cualquier lado de la fusión que toque un archivo que esté actualmente en conflicto.

```
$ git log --oneline --left-right --merge
< 694971d update phrase to hola world
> c3fffff1 changed text to hello mundo
```

En su lugar, si corremos eso con la opción `-p` obtendremos sólo los diffs del archivo que terminó en conflicto. Esto puede ser **bastante** útil, al darle rápidamente el contexto que necesita para ayudarle a entender por qué algo crea problemas y cómo resolverlo de una forma más inteligente.

Formato Diff Combinado

Dado que las etapas de Git clasifican los resultados que tienen éxito, cuando corre `git diff` mientras está en un estado de conflicto de fusión, sólo puede obtener lo que está actualmente en conflicto. Esto puede ser útil para ver lo que todavía debe resolver.

Cuando corre directamente `git diff` después de un conflicto de fusión, le dará la información en un formato de salida diff bastante único.

```
$ 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()
```

El formato es llamado “Diff combinado” y proporciona dos columnas de datos al lado de cada línea. La primera columna muestra si esa línea es diferente (añadida o removida) entre la rama “ours” y el archivo en su directorio de trabajo, y la segunda columna hace lo mismo entre la rama “theirs” y la copia de su directorio de trabajo.

Así que en ese ejemplo se puede observar que las líneas <<<<< y >>>>> están en la copia de trabajo, pero no en ningún lado de la fusión. Esto tiene sentido porque la herramienta de fusión las mantiene ahí para nuestro contexto, pero se espera que las removamos.

Si resolvemos el conflicto y corremos `git diff` de nuevo, veremos la misma cosa, pero es un poco más útil.

```
$ 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 mundo'
- puts 'hello mundo'
++ puts 'hola mundo'
end

hello()
```

Esto muestra que “hola mundo” estaba de nuestro lado, pero no en la copia de trabajo, que “hello mundo” estaba en el lado de ellos, pero no en la copia de trabajo y finalmente que “hola mundo” no estaba en ningún lado, sin embargo está ahora en la copia de trabajo. Esto puede ser útil para revisar antes de comprometer la resolución.

También se puede obtener desde el `git log` para cualquier fusión después de realizada, para ver cómo algo se resolvió luego de dicha fusión. Git dará salida a este formato si se puede correr `git show` en un compromiso de fusión, o si se añade la opción `--cc` a un `git log -p` (el cual por defecto solo muestra parches para compromisos no fusionados).

```

$ 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 mundo'
- puts 'hello mundo'
++ puts 'hola mundo'
end

hello()

```

Deshaciendo Fusiones

Ahora que ya conoce como crear un “merge commit” (compromiso de fusión), probablemente haya creado algunos por error. Una de las ventajas de trabajar con Git es que está bien cometer errores, porque es posible y, en muchos casos, es fácil solucionarlos.

Los compromisos de fusión no son diferentes. Digamos que comenzó a trabajar en una rama temática accidentalmente fusionada en una rama `master`, y ahora el historial de compromiso se ve así:

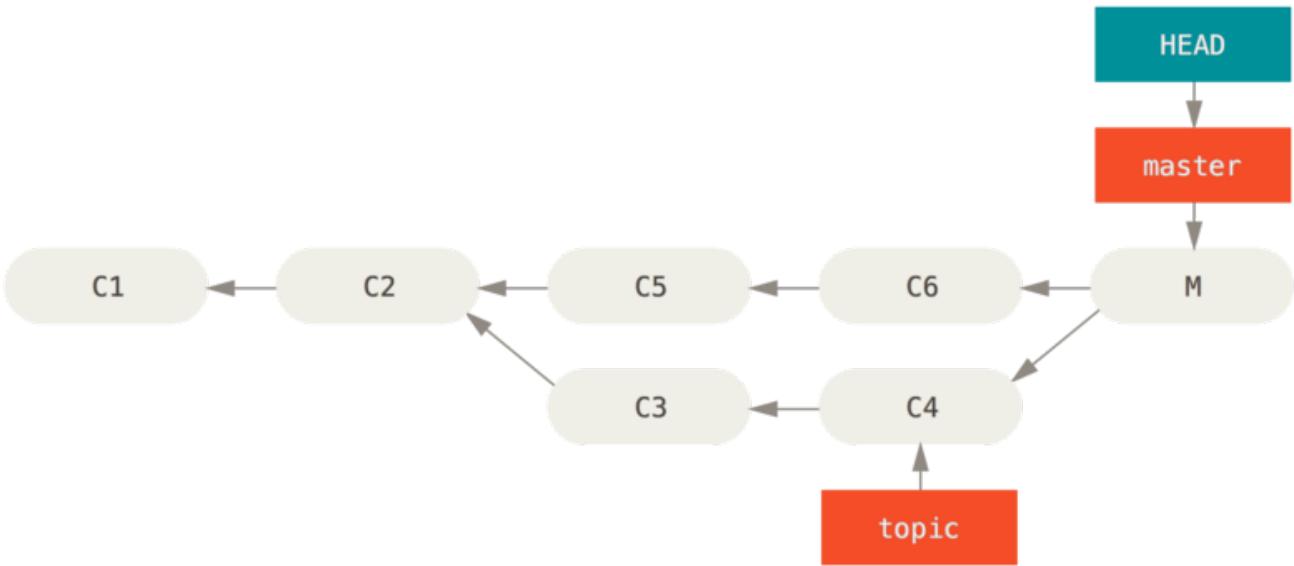


Figura 138. Accidental merge commit

Existen dos formas de abordar este problema, dependiendo de cuál es el resultado que desea.

Solucionar las referencias

Si el compromiso de fusión no deseado solo existe en su repositorio local, la mejor y más fácil solución es mover las ramas para que así apunten a dónde quiere que lo hagan. En la mayoría de los casos si sigue al errante `git merge` con `git reset --hard HEAD~`, esto restablecerá los punteros de la rama, haciendo que se vea así:

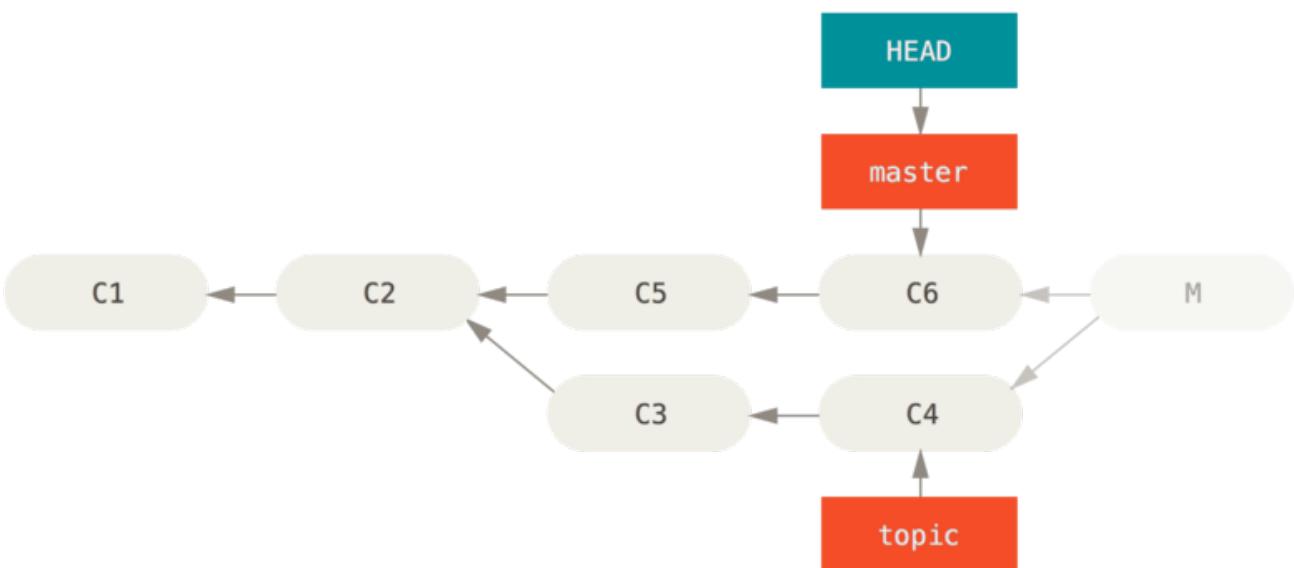


Figura 139. History after `git reset --hard HEAD~`

Ya vimos `reset` de nuevo en [Reiniciar Desmitificado](#), así que no debería ser muy difícil averiguar lo que está sucediendo. Aquí un repaso rápido: `reset --hard` usualmente va a través de tres pasos:

1. Mover los puntos de la rama HEAD. En este caso, se quiere mover la `principal`a donde se encontraba antes el compromiso de fusión ('C6).`

2. Hacer que el índice parezca HEAD.
3. Hacer que el directorio de trabajo parezca el índice.

La desventaja de este enfoque es que se reescribirá el historial, lo cual puede ser problemático con un depósito compartido. Revise [Los Peligros de Reorganizar](#) para saber más de lo que puede suceder; la versión corta es que, si otras personas tienen los compromisos que está reescribiendo, probablemente debería evitar [resetear](#). Este enfoque tampoco funcionará si cualquiera de los otros compromisos han sido creados desde la fusión; mover los refs efectivamente perdería esos cambios.

Revertir el compromiso

Si mover los punteros de la rama alrededor no funciona para su caso, Git le proporciona la opción de hacer un compromiso (“commit”) nuevo que deshace todos los cambios de uno ya existente. Git llama a esta operación un “revert”, y en este escenario en particular, ha invocado algo así:

```
$ git revert -m 1 HEAD
[master b1d8379] Revert "Merge branch 'topic'"
```

La bandera `-m 1` indica cuál padre es el “mainline” y debería ser mantenido. Cuando se invoque la fusión en el [HEAD \(git merge topic\)](#), el nuevo compromiso tiene dos padres: el primero es [HEAD \(C6\)](#), y el segundo es la punta de la rama siendo fusionada en [\(C4\)](#). En este caso, se quiere deshacer todos los cambios introducidos por el fusionamiento en el parent #2 ([C4](#)), pero manteniendo todo el contenido del parent #1 ([C6](#)).

El historial con el compromiso revertido se ve así:

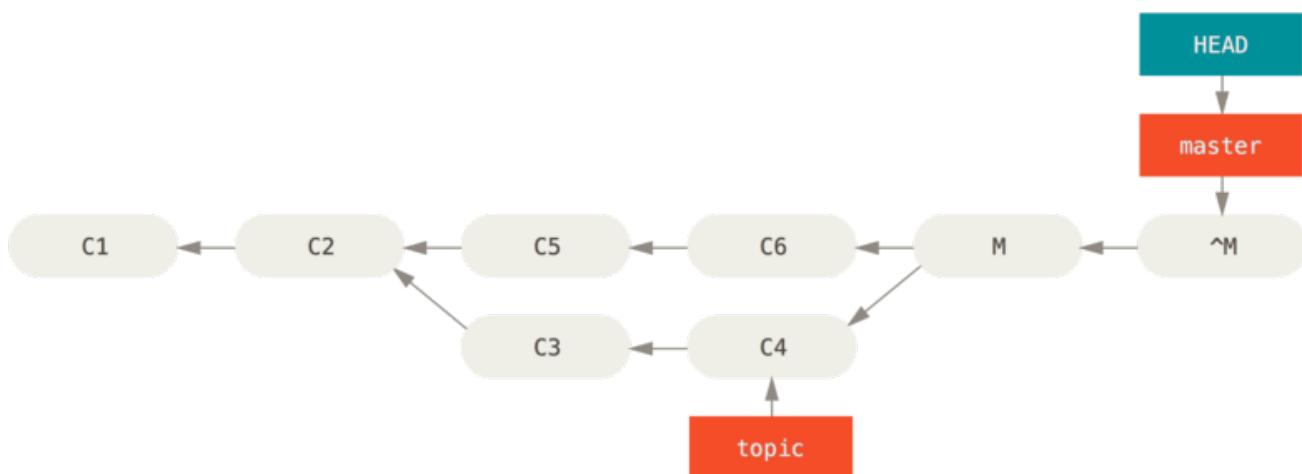


Figura 140. History after [git revert -m 1](#)

El nuevo compromiso [^M](#) tiene exactamente los mismos contenidos que [C6](#), así que comenzando desde aquí es como si la fusión nunca hubiese sucedido, excepto que ahora los no fusionados compromisos están todavía en [HEAD's history](#). Git se confundirá si intenta fusionar la rama [temática](#) en la rama [master](#):

```
$ git merge topic  
Already up-to-date.
```

No hay nada en `topic` que no sea ya alcanzable para la `master`. Que es peor, si añade trabajo a `topic` y fusiona otra vez, Git solo traerá los cambios desde la fusión revertida:

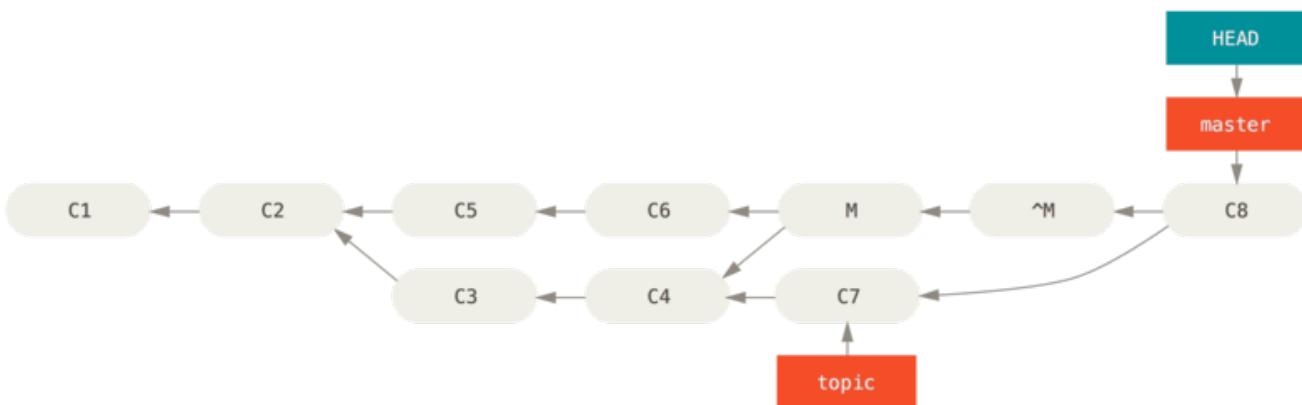


Figura 141. History with a bad merge

La mejor forma de evitar esto es deshacer la fusión original, dado que ahora se quiere traer los cambios que fueron revertidos, **luego** crear un nuevo compromiso de fusión:

```
$ git revert ^M  
[master 09f0126] Revert "Revert "Merge branch 'topic'""  
$ git merge topic
```

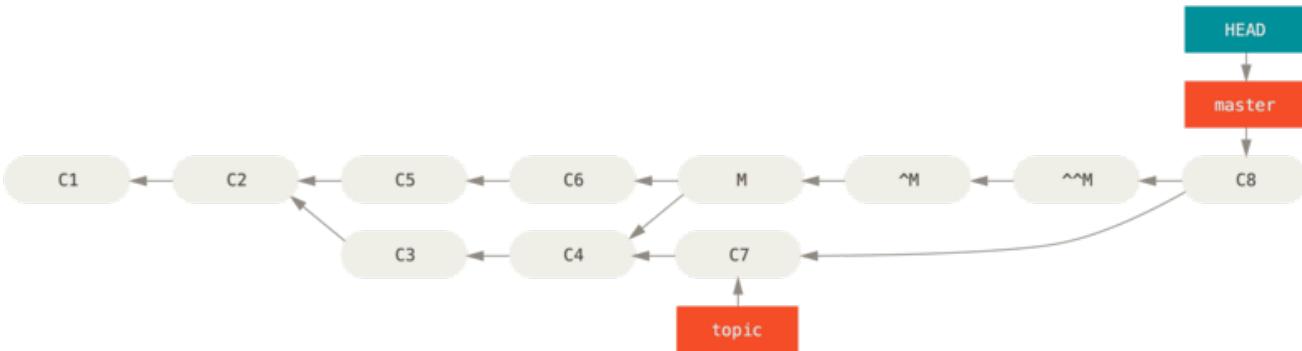


Figura 142. History after re-merging a reverted merge

En este ejemplo, `M` y `^M` se cancelan. Efectivamente `^^M` se fusiona en los cambios desde `C3` y `C4`, y `C8` se fusiona en los cambios desde `C7`, así que ahora `topic` está completamente fusionado.

Otros Tipos de Fusiones

Hasta hora ya cubrimos la fusión normal de dos ramas, normalmente manejado con lo que es llamado la estrategia de fusión “recursive”. Sin embargo, hay otras formas de fusionar a las ramas. Cubriremos algunas de ellas rápidamente.

Nuestra o Su preferencia

Primero que nada, hay otra cosa útil que podemos hacer con el modo de fusión “recursive”. Ya vimos las opciones `ignore-all-space` e `ignore-space-change` las cuales son pasadas con un `-X`, pero también le podemos decir a Git que favorezca un lado u otro cuando observe un conflicto.

Por defecto, cuando Git ve un conflicto entre dos ramas siendo fusionadas, añadirá marcadores de conflicto de fusión a los códigos, marcará el archivo como conflictivo y le dejará resolverlo. Si prefiere que Git simplemente escoja un lado específico e ignore el otro, en lugar de dejarle manualmente fusionar el conflicto, puede pasar el comando de fusión, ya sea on un `-Xours` o `-Xtheirs`.

Si Git ve esto, no añadirá marcadores de conflicto. Cualquier diferencia que pueda ser fusionable, se fusionará. Cualquier diferencia que entre en conflicto, él simplemente escogerá el lado que especifique en su totalidad, incluyendo los archivos binarios.

Si volvemos al ejemplo de “hello world” que estábamos utilizando antes, podemos ver que el fusionamiento en nuestra rama causa conflicto.

```
$ 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.
```

Sin embargo, si lo corremos con `-Xours` o `-Xtheirs` no lo causa.

```
$ 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
```

En este caso, en lugar de obtener marcadores de conflicto en el archivo con “hello mundo” en un lado y “hola world” en el otro, simplemente escogerá “hola world”. Sin embargo, todos los cambios no conflictivos en esa rama se fusionaron exitosamente.

Esta opción también puede ser trasmisida al comando `git merge-file` que vimos antes al correr algo como esto `git merge-file --ours` para archivos de fusión individuales.

Si quiere realizar algo así, pero Git no ha intentado siquiera fusionar cambios desde el otro lado, hay una opción más draconiana, la cual es la estrategia de fusión “ours” *merge strategy*. *Esto es diferente de la opción de fusión recursiva “ours” recursive merge_option.*

Esto básicamente hace una fusión falsa. Registrará un nuevo compromiso de fusión con

ambas ramas como padres, pero ni siquiera mirará a la rama que está fusionando. Simplemente registrará como el resultado de la fusión el código exacto en su rama actual.

```
$ git merge -s ours mundo
Merge made by the 'ours' strategy.
$ git diff HEAD HEAD~
$
```

Puede observar que no hay diferencia entre la rama en la que estábamos y el resultado de la fusión.

Esto a menudo puede ser útil para, básicamente, engañar a Git y que piense que una rama ya ha sido fusionada cuando se hace una fusión más adelante. Por ejemplo, decir que ha ramificado una rama de “release” y ha hecho un poco de trabajo que querrá fusionar de vuelta en su rama “master” en algún punto. Mientras tanto, algunos arreglos de fallos en la “master” necesitan ser adaptados en la rama de `release`. Se puede fusionar la rama “bugfix” en la de `release` y también `merge -s ours`, la misma rama en la principal (a pesar de que el arreglo ya se encuentre ahí). Así que, más tarde cuando fuseone la de lanzamiento otra vez, no hay conflictos del “bugfix”.

Convergencia de Subárbol

La idea de la convergencia de subárboles es que usted tiene dos proyectos, de los cuales uno lleva un subdirectorio del otro y viceversa. Cuando especifica una convergencia de subárbol, Git suele ser lo suficientemente inteligente para comprender que uno es un subárbol del otro y convergerá apropiadamente.

Veremos un ejemplo donde se añade un proyecto separado a un proyecto existente y luego se converge el código del segundo dentro de un subdirectorio del primero.

Primero, añadiremos la aplicación Rack a nuestro proyecto. Añadiremos el proyecto Rack como referencia remota en nuestro propio proyecto y luego lo colocaremos en su propia 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"

```

Ahora tenemos la raíz del proyecto Rack en nuestro branch `rack_branch` y nuestro proyecto en el branch `master`. Si verifica uno y luego el otro, puede observar que tienen diferentes raíces de proyecto:

```

$ ls
AUTHORS      KNOWN-ISSUES  Rakefile      contrib      lib
COPYING      README        bin          example      test
$ git checkout master
Switched to branch "master"
$ ls
README

```

Este concepto es algo extraño. No todas las *branches* en su repositorio tendrán que ser *branches* del mismo proyecto como tal. No es común, porque rara vez es de ayuda, pero es fácil que los *branches* contengan historias completamente diferentes.

En este caso, queremos integrar el proyecto Rack a nuestro proyecto `master` como un subdirectorio. Podemos hacer eso en Git con `git read-tree`. Aprenderá más sobre `read-tree` y sus amigos en [Los entresijos internos de Git](#), pero por ahora sepia que éste interpreta el árbol raíz de una *branch* en su *'área de staging'* y *'directorio de trabajo'*. Sólo cambiamos de vuelta a su *branch* `master`, e integramos la *branch* `rack_branch` al subdirectorio `rack` de nuestra *branch* `master` de nuestro proyecto principal:

```
$ git read-tree --prefix=rack/ -u rack_branch
```

Cuando hacemos “commit”, parece que tenemos todos los archivos Rack bajo ese subdirectorio - como si los hubiéramos copiado de un tarball. Lo interesante es que podemos fácilmente converger cambios de una de las *branches* a la otra. Entonces, si el proyecto Rack se actualiza, podemos atraer cambios río arriba alternando a esa *branch* e