

THE EXPERT'S VOICE® IN SOFTWARE DEVELOPMENT

# Pro Git

*Everything you need to know about  
the Git distributed source control tool*



Scott Chacon

*Foreword by Junio C Hamano, Git project leader*

Apress®

# 1.1 Empezando - Acerca del control de versiones

## Acerca del control de versiones

¿Qué es el control de versiones, y por qué debería importarte? El control de versiones es un sistema que registra los cambios realizados sobre un archivo o conjunto de archivos a lo largo del tiempo, de modo que puedas recuperar versiones específicas más adelante. A pesar de que los ejemplos de este libro muestran código fuente como archivos bajo control de versiones, en realidad cualquier tipo de archivo que encuentres en un ordenador puede ponerse bajo control de versiones.

Si eres diseñador gráfico o web, y quieres mantener cada versión de una imagen o diseño (algo que sin duda quieres), un sistema de control de versiones (Version Control System o VCS en inglés) es una elección muy sabia. Te permite revertir archivos a un estado anterior, revertir el proyecto entero a un estado anterior, comparar cambios a lo largo del tiempo, ver quién modificó por última vez algo que puede estar causando un problema, quién introdujo un error y cuándo, y mucho más. Usar un VCS también significa generalmente que si fastidias o pierdes archivos, puedes recuperarlos fácilmente. Además, obtienes todos estos beneficios a un coste muy bajo.

## Sistemas de control de versiones locales

Un método de control de versiones usado por mucha gente es copiar los archivos a otro directorio (quizás indicando la fecha y hora en que lo hicieron, si son avispados). Este enfoque es muy común porque es muy simple, pero también tremendamente 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 hacer frente a este problema, los programadores desarrollaron hace tiempo VCSs locales que contenían una simple base de datos en la que se llevaba registro de todos los cambios realizados sobre los archivos (véase Figura 1-1).

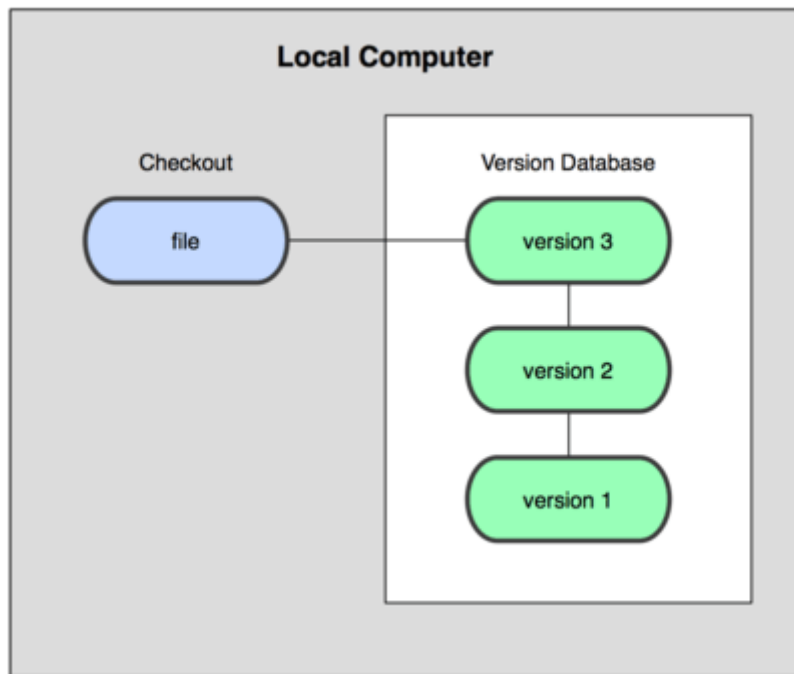


Figura 1-1. Diagrama de control de versiones local.

Una de las herramientas de control de versiones más popular fue un sistema llamado rcs, que todavía podemos encontrar en muchos de los ordenadores actuales. Hasta el famoso sistema operativo Mac OS X incluye el comando rcs cuando instalas las herramientas de desarrollo. Esta herramienta funciona básicamente guardando conjuntos de parches (es decir, las diferencias entre archivos) de una versión a otra en un formato especial en disco; puede entonces recrear cómo era un archivo en cualquier momento sumando los distintos parches.

### Sistemas de control de versiones centralizados

El siguiente gran problema que se encuentra la gente es que necesitan colaborar con desarrolladores en otros sistemas. Para solventar este problema, se desarrollaron los sistemas de control de versiones centralizados (Centralized Version Control Systems o CVCSs en inglés). 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. Durante muchos años éste ha sido el estándar para el control de versiones (véase Figura 1-2).

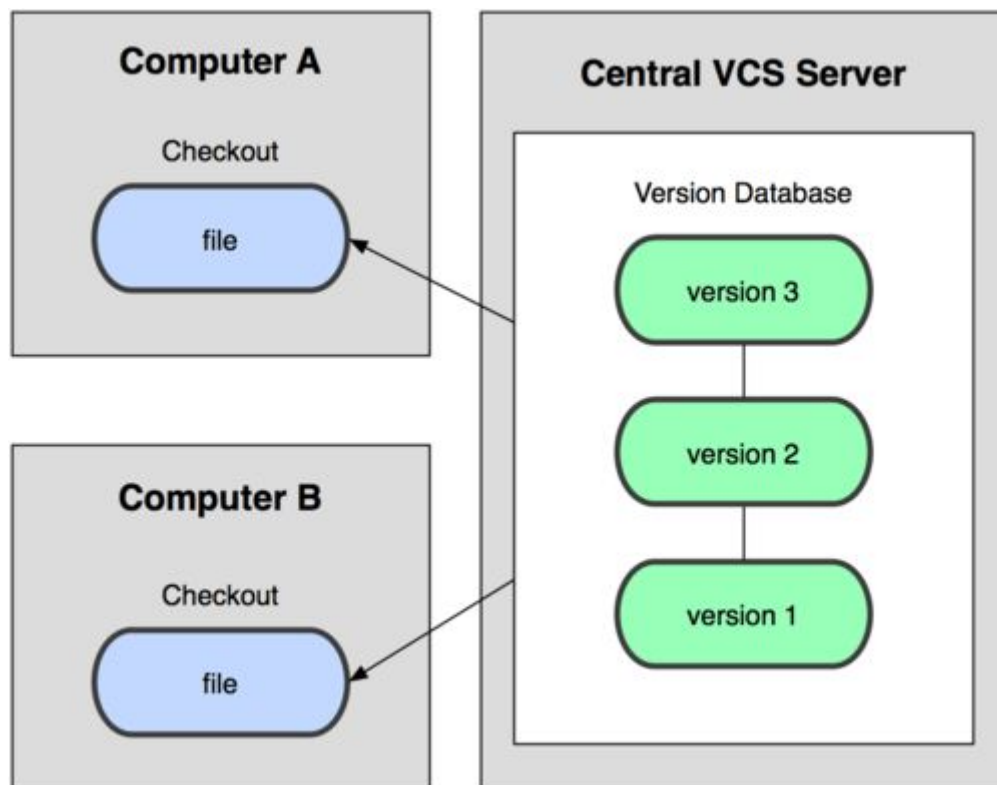


Figura 1-2. Diagrama de control de versiones centralizado.

Esta configuración ofrece muchas ventajas, especialmente frente a VCSs locales. Por ejemplo, todo el mundo puede saber (hasta cierto punto) en qué están trabajando los otros colaboradores del proyecto. Los administradores tienen control detallado de qué puede hacer cada uno; 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 puede colaborar o guardar cambios versionados de aquello en que están trabajando. Si el disco duro en el que se encuentra la base de datos central se corrompe, y no se han llevado copias de seguridad adecuadamente, pierdes absolutamente todo —toda la historia del proyecto salvo aquellas instantáneas que la gente pueda tener en sus máquinas locales. Los VCSs locales sufren de este mismo problema— cuando tienes toda la historia del proyecto en un único lugar, te arriesgas a perderlo todo.

### Sistemas de control de versiones distribuidos

Es aquí donde entran los sistemas de control de versiones distribuidos (Distributed Version Control Systems o DVCSs en inglés). En un DVCS (como Git, Mercurial, Bazaar o Darcs), los clientes no sólo descargan la última instantánea de los archivos: replican completamente el

repositorio. Así, si un servidor muere, y estos sistemas estaban colaborando a través de él, cualquiera de los repositorios de los clientes puede copiarse en el servidor para restaurarlo. Cada vez que se descarga una instantánea, en realidad se hace una copia de seguridad completa de todos los datos (véase Figura 1-3).

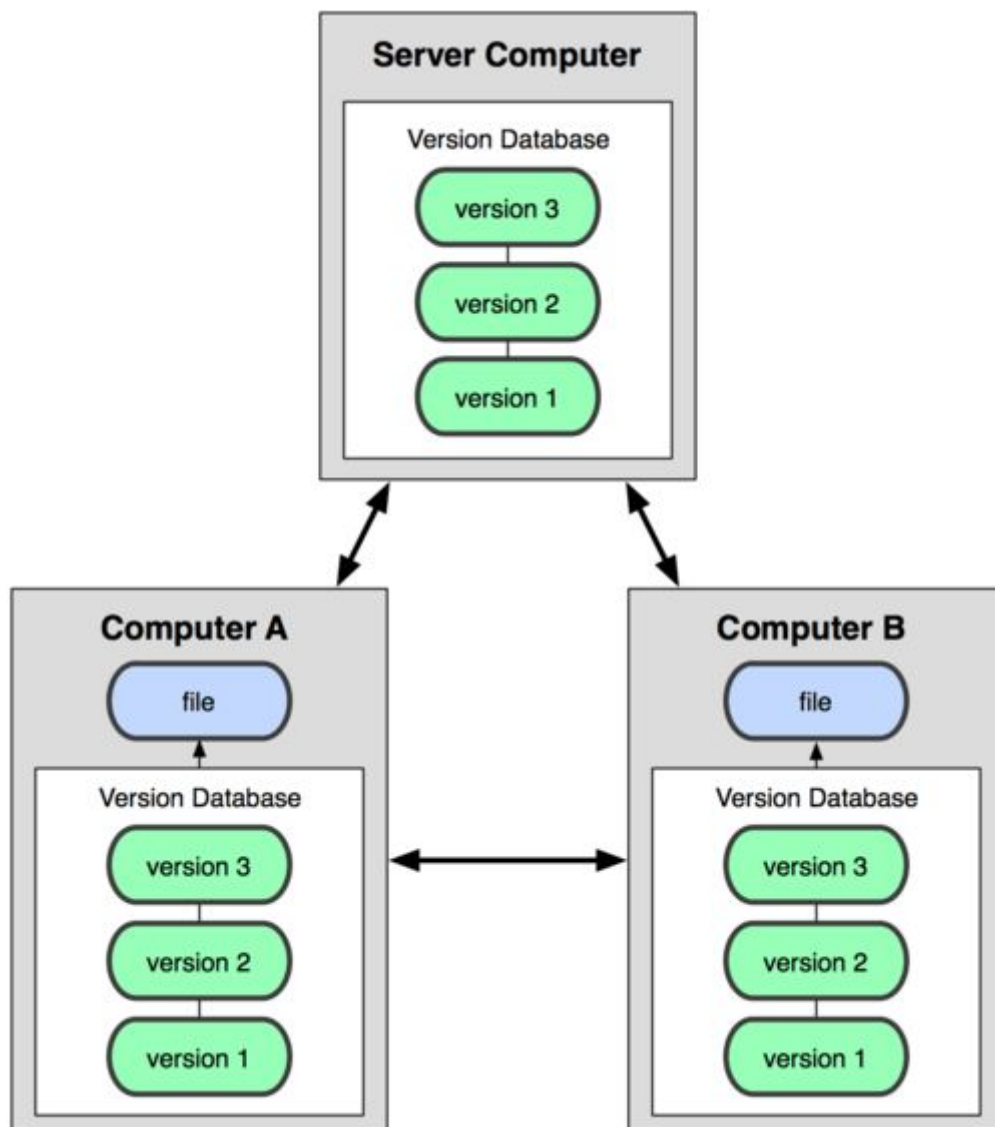


Figura 1-3. Diagrama de control de versiones distribuido.

Es más, muchos de estos sistemas se las arreglan bastante bien teniendo varios repositorios con los que trabajar, por lo que puedes colaborar con distintos grupos de gente simultáneamente dentro del mismo proyecto. Esto te permite establecer varios flujos de trabajo que no son posibles en sistemas centralizados, como pueden ser los modelos jerárquicos.

# 1.2 Empezando - Una breve historia de Git

## Una breve historia de Git

Como muchas de las grandes cosas en esta vida, Git comenzó con un poco de destrucción creativa y encendida polémica. El núcleo de Linux es un proyecto de software de código abierto con un alcance bastante grande. Durante la mayor parte del mantenimiento del núcleo de Linux (1991-2002), los cambios en el software se pasaron en forma de parches y archivos. En 2002, el proyecto del núcleo de Linux empezó a usar un DVCS propietario llamado BitKeeper.

En 2005, la relación entre la comunidad que desarrollaba el núcleo de Linux y la compañía que desarrollaba BitKeeper se vino abajo, y la herramienta dejó de ser ofrecida gratuitamente. 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 durante el uso de BitKeeper. Algunos de los objetivos del nuevo sistema fueron los siguientes:

- Velocidad
- Diseño sencillo
- Fuerte apoyo al desarrollo no lineal (miles de ramas paralelas)
- Completamente distribuido
- Capaz de manejar grandes proyectos (como el núcleo de Linux) de manera eficiente (velocidad y tamaño de los datos)

Desde su nacimiento en 2005, Git ha evolucionado y madurado para ser fácil de usar y aún conservar estas cualidades iniciales. Es tremendamente rápido, muy eficiente con grandes proyectos, y tiene un increíble sistema de ramificación (branching) para desarrollo no lineal (véase el Capítulo 3).

# 1.3 Empezando - Fundamentos de Git

## Fundamentos de Git

Entonces, ¿qué es Git en pocas palabras? Es muy importante asimilar esta sección, porque si entiendes lo que es Git y los fundamentos de cómo funciona, probablemente te sea mucho más fácil usar Git de manera eficaz. A medida que aprendas Git, intenta olvidar todo lo que puedas saber sobre otros VCSs, como Subversion y Perforce; hacerlo te ayudará a evitar confusiones sutiles a la hora de utilizar la herramienta. Git almacena y modela la información de forma muy diferente a esos otros sistemas, a pesar de que su interfaz sea bastante similar; comprender esas diferencias evitará que te confundas a la hora de usarlo.

### Instantáneas, no diferencias

La principal diferencia entre Git y cualquier otro VCS (Subversion y compañía incluidos) es cómo Git modela sus datos. Conceptualmente, la mayoría de los demás sistemas almacenan la información como una lista de cambios en los archivos. Estos sistemas (CVS, Subversion, Perforce, Bazaar, etc.) modelan la información que almacenan como un conjunto de archivos y las modificaciones hechas sobre cada uno de ellos a lo largo del tiempo, como ilustra la Figura 1-4.

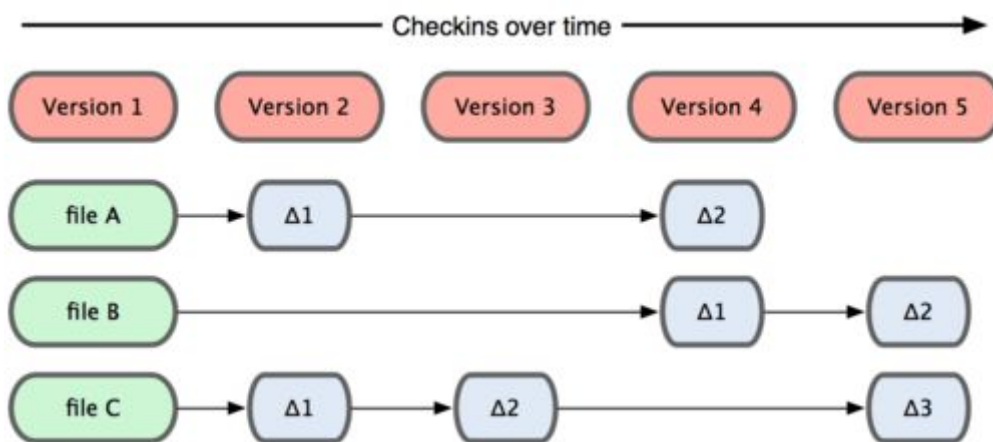


Figura 1-4. Otros sistemas tienden a almacenar los datos como cambios de cada archivo respecto a una versión base.

Git no modela ni almacena sus datos de este modo. En cambio, Git modela sus datos más como un conjunto de instantáneas de un mini sistema de archivos. Cada vez que confirmas un cambio, o guardas el estado de tu proyecto en Git, él básicamente hace una foto del aspecto de todos tus archivos en ese momento, y guarda una referencia a esa instantánea. Para ser eficiente, si los



archivos no se han modificado, Git no almacena el archivo de nuevo, sólo un enlace al archivo anterior idéntico que ya tiene almacenado. Git modela sus datos más como en la Figura 1-5.

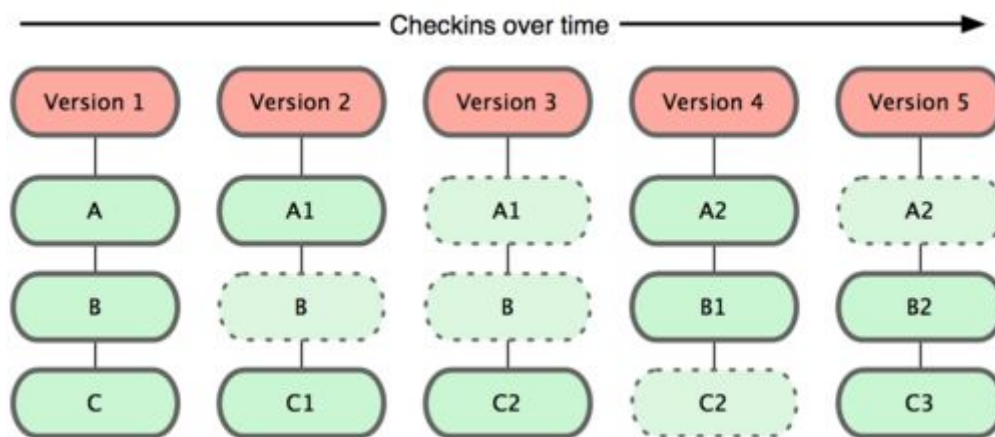


Figura 1-5. Git almacena la información como instantáneas del proyecto a lo largo del tiempo. Esta es una distinción importante entre Git y prácticamente todos los demás VCSs. Hace que Git reconsidere casi todos los aspectos del control de versiones que muchos de los demás sistemas copiaron de la generación anterior. Esto hace que Git se parezca más a un mini sistema de archivos con algunas herramientas tremendamente potentes construidas sobre él, que a un VCS. Exploraremos algunos de los beneficios que obtienes al modelar tus datos de esta manera cuando veamos ramificaciones (branching) en Git en el Capítulo 3.

### Casi cualquier operación es local

La mayoría de las operaciones en Git sólo necesitan archivos y recursos locales para operar. Por lo general no se necesita información de ningún otro ordenador de tu red. Si estás acostumbrado a un CVCS donde la mayoría de las operaciones tienen esa sobrecarga 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. Como 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 salir 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 al instante. Si quieres ver los cambios introducidos en un archivo entre la versión actual y la de hace un mes, Git puede buscar el archivo hace un mes y hacer un cálculo de diferencias localmente, en lugar de tener que pedirle a un servidor remoto que lo haga, u obtener una versión antigua desde la red y hacerlo de manera local.

Esto también significa que hay muy poco que no puedas 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 doloroso. En Perforce, por ejemplo, no puedes hacer mucho cuando no estás conectado al servidor; y 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 lo detecte.

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

**24b9da6552252987aa493b52f8696cd6d3b00373**

Verás estos valores hash por todos lados en Git, ya que los usa con mucha frecuencia. De hecho, Git guarda todo no por nombre de archivo, sino por el valor hash de sus contenidos.

## **Git generalmente sólo 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 deshacer, 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 instantánea en Git, es muy difícil de perder, especialmente si envías (push) 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 fastidiar gravemente las cosas. Para un análisis más exhaustivo de cómo almacena Git su información y cómo puedes recuperar datos aparentemente perdidos, ver Capítulo 9.

## **Los tres estados**

Ahora presta atención. Esto es lo más importante a recordar acerca de Git si quieres 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).

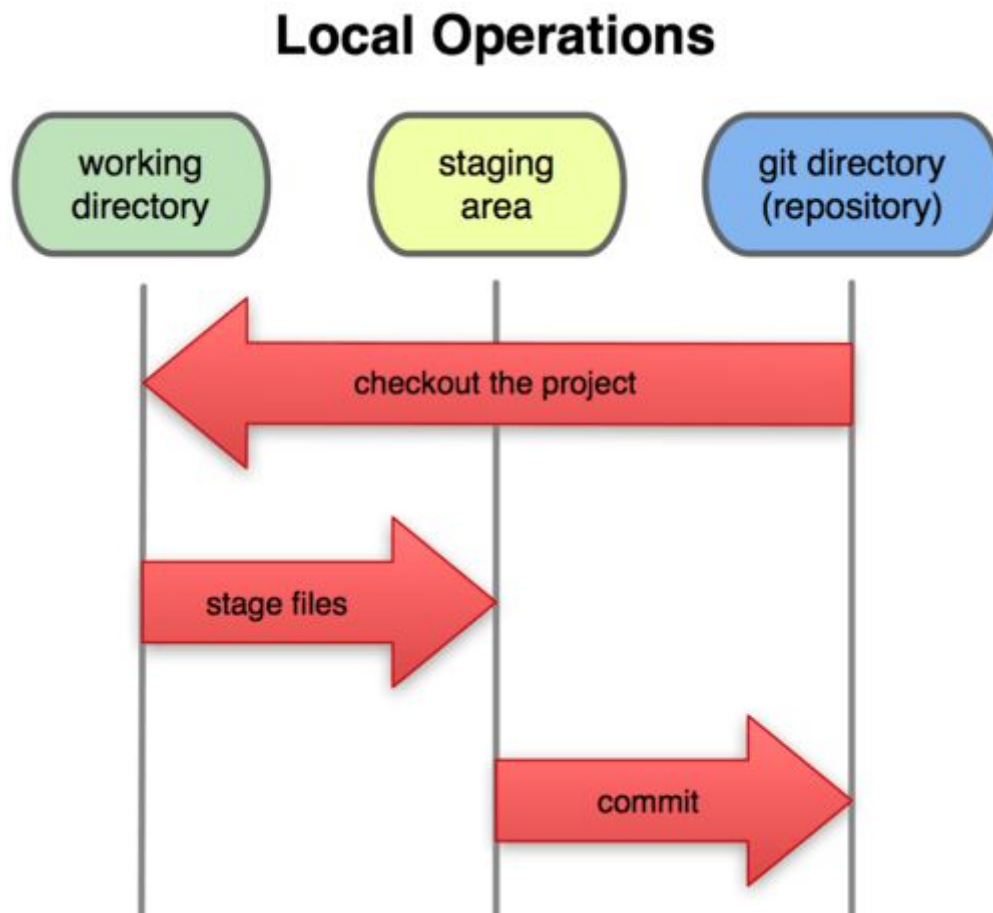


Figura 1-6. Directorio de trabajo, área de preparación y directorio de Git. El directorio de Git es donde Git almacena los metadatos y la base de datos de objetos para tu proyecto. Es la parte más importante de Git, y es lo que se copia cuando clonas un repositorio desde otro ordenador.

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 sencillo 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, 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 esas instantáneas de manera permanente en tu directorio de Git.

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

# 1.5 Empezando - Configurando Git por primera vez

## Configurando Git por primera vez

Ahora que tienes Git en tu sistema, querrás hacer algunas cosas para personalizar tu entorno de Git. Sólo es necesario hacer estas cosas una vez; 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:

- 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.
- Archivo **~/.gitconfig** file: Específico a tu usuario. Puedes hacer que Git lea y escriba específicamente en este archivo pasando la opción **--global**.
- Archivo config en el directorio de Git (es decir, **.git/config**) del repositorio que estés utilizando actualmente: Específico a ese repositorio. 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** (**%USERPROFILE%** in Windows' environment), que es **C:\Documents and Settings\%USER** para la mayoría de usuarios, dependiendo de la versión (**\$USER** es **%USERNAME%** en el entorno Windows). También busca en el directorio **/etc/gitconfig**, aunque esta ruta es relativa a la raíz MSys, que es donde quiera que decidieses instalar Git en tu sistema Windows cuando ejecutaste el instalador.

## Tu identidad

Lo primero que deberías hacer cuando instalas Git es establecer tu nombre de usuario y dirección de correo electrónico. Esto es importante porque las confirmaciones de cambios (commits) en 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.

## **Tu editor**

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

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

## **Tu herramienta de diferencias**

Otra opción útil que puede que quieras configurar es la herramienta de diferencias por defecto, usada para resolver conflictos de unión (merge). Digamos que quieres usar vimdiff:

```
$ git config --global merge.tool vimdiff
```

Git acepta kdiff3, tkdiff, meld, xxdiff, emerge, vimdiff, gvimdiff, ecmerge, y opendiff como herramientas válidas. También puedes configurar la herramienta que tú quieras; véase el Capítulo 7 para más información sobre cómo hacerlo.

## **Comprobando tu configuración**

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

```
$ git config --list
user.name=Scott Chacon
user.email=schacon@gmail.com
color.status=auto
color.branch=auto
color.interactive=auto
color.diff=auto
```

...

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

También puedes comprobar qué valor cree Git que tiene una clave específica ejecutando `git config {clave}`:

```
$ git config user.name
Scott Chacon
```

## 1.6 Empezando - Obteniendo ayuda

### Obteniendo ayuda

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

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

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

```
$ git help config
```

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

# Chapter 2

## Fundamentos de Git

Si sólo puedes leer un capítulo para empezar a trabajar con Git, es éste. Este capítulo cubre todos los comandos básicos que necesitas para hacer la gran mayoría de las cosas a las que vas a dedicar tu tiempo en Git. Al final del capítulo, deberías 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 deshacer 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.

## 2.1 Fundamentos de Git - Obteniendo un repositorio Git

### Obteniendo un repositorio Git

Puedes obtener un proyecto Git de dos maneras. La primera toma un proyecto o directorio existente y lo importa en Git. La segunda clona un repositorio Git existente desde otro servidor.

#### Inicializando un repositorio en un directorio existente

Si estás empezando el seguimiento en Git de un proyecto existente, necesitas ir al directorio del proyecto y escribir:

```
$ git init
```

Esto crea un nuevo subdirectorio llamado `.git` que contiene todos los archivos necesarios del repositorio —un esqueleto de un repositorio Git. Todavía no hay nada en tu proyecto que esté bajo seguimiento. (Véase el Capítulo 9 para obtener más información sobre qué archivos están contenidos en el directorio `.git` que acabas de crear.)



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 `commit` para confirmar los cambios:

```
$ git add *.c
$ git add README
$ git commit -m 'versión inicial del proyecto'
```

Veremos lo que hacen estos comandos dentro de un minuto. En este momento, tienes un repositorio 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` y no `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 descargado cuando ejecutas `git clone`. De hecho, si el disco de tu servidor se corrompe, puedes usar cualquiera de los clones en cualquiera de los clientes para devolver al servidor al estado en el que estaba cuando fue clonado (puede que pierdas algunos *hooks* del lado del servidor y demás, pero toda la información versionada estaría ahí —véase el Capítulo 4 para más detalles—).

Puedes clonar un repositorio con `git clone [url]`. Por ejemplo, si quieres clonar la librería Ruby llamada Grit, harías algo así:

```
$ git clone git://github.com/schacon/grit.git
```

Esto crea un directorio llamado "grit", 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 nuevo directorio `grit`, 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 `grit`, puedes especificarlo con la siguiente opción de línea de comandos:

```
$ git clone git://github.com/schacon/grit.git mygrit
```

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

Git te permite usar distintos protocolos de transferencia. El ejemplo anterior usa el protocolo `git://`, pero también te puedes encontrar con `http(s)://` o `usuario@servidor:/ruta.git`, que utiliza el protocolo de transferencia SSH. En el Capítulo 4 se introducirán todas las opciones disponibles a la hora de configurar el acceso a tu repositorio Git, y las ventajas e inconvenientes de cada una.

## 2.2 Fundamentos de Git - Guardando cambios en el repositorio

### Guardando cambios en el repositorio

Tienes un repositorio Git completo, y una copia de trabajo de los archivos de ese proyecto. Necesitas hacer algunos cambios, y confirmar instantáneas de esos cambios a tu repositorio cada vez que el proyecto alcance un estado que desees grabar.

Recuerda que cada archivo de tu directorio de trabajo puede estar en uno de estos dos estados: bajo seguimiento (tracked), o sin seguimiento (untracked). Los archivos bajo seguimiento son aquellos que existían en la última instantánea; pueden estar sin modificaciones, modificados, o preparados. Los archivos sin seguimiento son todos los demás —cualquier archivo de tu directorio que no estuviese en tu última instantánea ni está en tu área de preparación—. La primera vez que clonas un repositorio, todos tus archivos estarán bajo seguimiento y sin modificaciones, ya que los acabas de copiar y no has modificado nada.

A medida que editas archivos, Git los ve como modificados, porque los has cambiado desde tu última confirmación. Preparas estos archivos modificados y luego confirmas todos los cambios que hayas preparado, y el ciclo se repite. Este proceso queda ilustrado en la Figura 2-1.

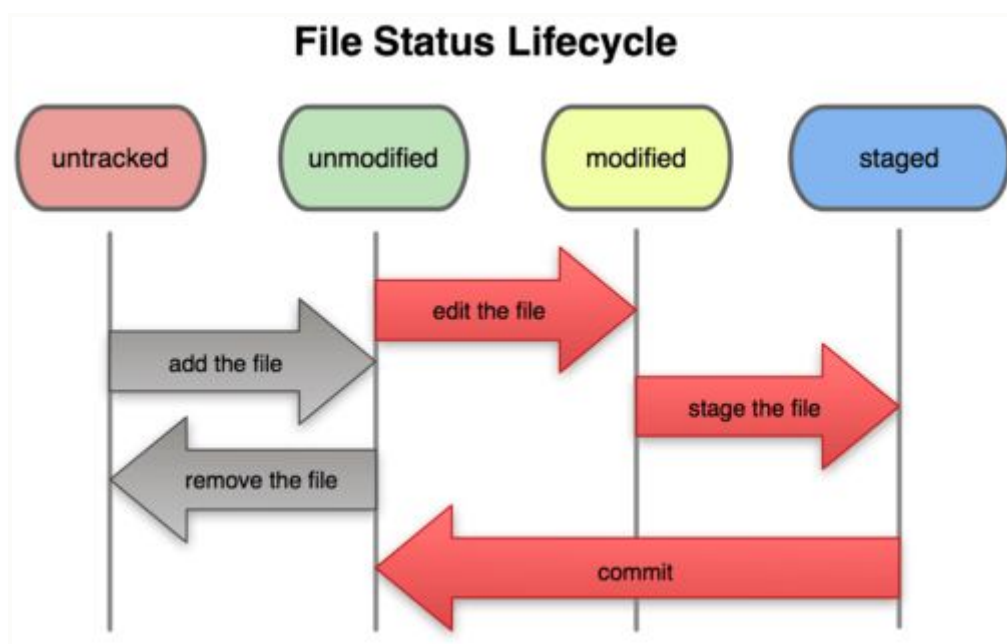


Figura 2-1. El ciclo de vida del estado de tus archivos.

## Comprobando el estado de tus archivos

Tu principal herramienta para determinar qué archivos están en qué estado es el comando **git status**. Si ejecutas este comando justo después de clonar un repositorio, deberías ver algo así:

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

Esto significa que tienes un directorio de trabajo limpio —en otras palabras, no tienes archivos bajo seguimiento y modificados—. Git tampoco ve ningún archivo que no esté bajo seguimiento, o estaría listado ahí. Por último, el comando te dice en qué rama estás. Por ahora, esa rama siempre es "master", que es la predeterminada. No te preocupes de eso por ahora, el siguiente capítulo tratará los temas de las ramas y las referencias en detalle.

Digamos que añades un nuevo archivo a tu proyecto, un sencillo archivo README. Si el archivo no existía y ejecutas **git status**, verás tus archivos sin seguimiento así:

```
$ vim 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 tu nuevo archivo README aparece bajo la cabecera “Archivos sin seguimiento” (“Untracked files”) de la salida del comando. Sin seguimiento significa básicamente que Git ve un archivo que no estaba en la instantánea anterior; Git no empezará a incluirlo en las confirmaciones de tus instantáneas hasta que se lo indiques explícitamente. Lo hace para que no incluyas accidentalmente archivos binarios generados u otros archivos que no tenías intención de incluir. Sí que quieres incluir el README, así que vamos a iniciar el seguimiento del archivo.

## Seguimiento de nuevos archivos

Para empezar el seguimiento de un nuevo archivo se usa el comando **git add**. Iniciaremos el seguimiento del archivo README ejecutando esto:

```
$ git add README
```

Si vuelves a ejecutar el comando **git status**, verás que tu README está ahora bajo seguimiento y preparado:

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

Puedes ver que está preparado porque aparece bajo la cabecera “Cambios a confirmar” (“Changes to be committed”). Si confirmas ahora, la versión del archivo en el momento de ejecutar **git add** será la que se incluya en la instantánea. Recordarás que cuando antes ejecutaste **git init**, seguidamente ejecutaste **git add (archivos)**. Esto era para iniciar el seguimiento de los archivos de tu directorio. El comando **git add** recibe la ruta de un archivo o de un directorio; si es un directorio, añade todos los archivos que contenga de manera recursiva.

## **Preparando archivos modificados**

Vamos a modificar un archivo que estuviese bajo seguimiento. Si modificas el archivo **benchmarks.rb** que estaba bajo seguimiento, y ejecutas el comando **status** de nuevo, verás algo así:

```
$ 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)
#
#   modified:   benchmarks.rb
#
```

El archivo `benchmarks.rb` aparece bajo la cabecera “Modificados pero no actualizados” (“Changes not staged for commit”) —esto significa que un archivo bajo seguimiento ha sido modificado en el directorio de trabajo, pero no ha sido preparado todavía—. Para prepararlo, ejecuta el comando `git add` (es un comando multiuso —puedes utilizarlo para empezar el seguimiento de archivos nuevos, para preparar archivos, y para otras cosas como marcar como resueltos archivos con conflictos de unión—). Ejecutamos `git add` para preparar el archivo `benchmarks.rb`, y volvemos a ejecutar `git status`:

```
$ git add benchmarks.rb
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   new file:   README
#   modified:   benchmarks.rb
#
```

Ambos archivos están ahora preparados y se incluirán en tu próxima confirmación. Supón que en este momento recuerdas que tenías que hacer una pequeña modificación en `benchmarks.rb` antes de confirmarlo. Lo vuelves abrir, haces ese pequeño cambio, y ya estás listo para confirmar. Sin embargo, si vuelves a ejecutar `git status` verás lo siguiente:

```
$ vim benchmarks.rb
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   new file:   README
#   modified:   benchmarks.rb
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#
#   modified:   benchmarks.rb
#
```

¿Pero qué...? Ahora benchmarks.rb aparece listado como preparado y como no preparado. ¿Cómo es posible? Resulta que Git prepara un archivo tal y como era en el momento de ejecutar el comando `git add`. Si haces `git commit` ahora, la versión de benchmarks.rb que se incluirá en la confirmación será la que fuese cuando ejecutaste el comando `git add`, no la versión que estás viendo ahora en tu directorio de trabajo. Si modificas un archivo después de haber ejecutado `git add`, tendrás que volver a ejecutar `git add` para preparar la última versión del archivo:

```
$ git add benchmarks.rb
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   new file:   README
#   modified:   benchmarks.rb
#
```

## Ignorando archivos

A menudo tendrás un tipo de archivos que no quieras que Git añada automáticamente o te muestre como no versionado. Suelen ser archivos generados automáticamente, como archivos de log, o archivos generados por tu compilador. Para estos casos puedes crear un archivo llamado `.gitignore`, en el que listas los patrones de nombres que deseas que sean ignorados. He aquí un archivo `.gitignore` de ejemplo:

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

La primera línea le dice a Git que ignore cualquier archivo cuyo nombre termine en `.o` o `.a` —archivos objeto que suelen ser producto de la compilación de código—. La segunda línea le dice a Git que ignore todos los archivos que terminan en tilde (`~`), usada por muchos editores de texto, como Emacs, para marcar archivos temporales. También puedes incluir directorios de log, temporales, documentación generada automáticamente, etc. Configurar un archivo `.gitignore` antes de empezar a trabajar suele ser una buena idea, para así no confirmar archivos que no quieres en tu repositorio Git.

Las reglas para los patrones que pueden ser incluidos en el archivo `.gitignore` son:



- Las líneas en blanco, o que comienzan por #, son ignoradas.
- Puedes usar patrones glob estándar.
- Puedes indicar un directorio añadiendo una barra hacia delante (/) al final.
- Puedes negar un patrón añadiendo una exclamación (!) al principio.

Los patrones glob son expresiones regulares simplificadas que pueden ser usadas por las shells.

Un asterisco (\*) reconoce cero o más caracteres; [abc] reconoce cualquier carácter de los especificados entre corchetes (en este caso, a, b o c); una interrogación (?) reconoce un único carácter; y caracteres entre corchetes separados por un guión ([0-9]) reconoce cualquier carácter entre ellos (en este caso, de 0 a 9).

He aquí otro ejemplo de archivo .gitignore:

```
# a comment – this is ignored
# no .a files
*.a
# but do track lib.a, even though you're ignoring .a files above
!lib.a
# only ignore the root TODO file, not subdir/TODO
/TODO
# ignore all files in the build/ directory
build/
# ignore doc/notes.txt, but not doc/server/arch.txt
doc/*.txt
# ignore all .txt files in the doc/ directory
doc/**/*.txt
```

El patrón \*/ está disponible en Git desde la versión 1.8.2.

## Viendo tus cambios preparados y no preparados

Si el comando `git status` es demasiado impreciso para ti —quieres saber exactamente lo que ha cambiado, no sólo qué archivos fueron modificados— puedes usar el comando `git diff`. Veremos `git diff` en más detalle después; pero probablemente lo usarás para responder estas dos preguntas: ¿qué has cambiado pero aún no has preparado?, y ¿qué has preparado y estás a punto de confirmar? Aunque `git status` responde esas preguntas de manera general, `git diff` te muestra exactamente las líneas añadidas y eliminadas —el parche, como si dijésemos.

Supongamos que quieres editar y preparar el archivo README otra vez, y luego editar el archivo benchmarks.rb sin prepararlo. Si ejecutas el comando `status`, de nuevo verás algo así:

```
$ 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)
#
#   modified:   benchmarks.rb
#
```

Para ver lo que has modificado pero aún no has preparado, escribe **git diff**:

```
$ git diff
diff --git a/benchmarks.rb b/benchmarks.rb
index 3cb747f..da65585 100644
--- a/benchmarks.rb
+++ b/benchmarks.rb
@@ -36,6 +36,10 @@ def main
   @commit.parents[0].parents[0].parents[0]
   end

+  run_code(x, 'commits 1') do
+    git.commits.size
+  end
+
   run_code(x, 'commits 2') do
     log = git.commits('master', 15)
     log.size
```

Ese comando compara lo que hay en tu directorio de trabajo con lo que hay en tu área de preparación. El resultado te indica los cambios que has hecho y que todavía no has preparado.

Si quieres ver los cambios que has preparado y que irán en tu próxima confirmación, puedes usar **git diff --cached**. (A partir de la versión 1.6.1 de Git, también puedes usar **git diff --staged**, que puede resultar más fácil de recordar.) Este comando compara tus cambios preparados con tu última confirmación:

```
$ git diff --cached
diff --git a/README b/README
new file mode 100644
index 0000000..03902a1
--- /dev/null
+++ b/README2
@@ -0,0 +1,5 @@
+grit
+ by Tom Preston-Werner, Chris Wanstrath
+ http://github.com/mojombo/grit
+
+Grit is a Ruby library for extracting information from a Git repository
```

Es importante indicar que `git diff` por sí solo no muestra todos los cambios hechos desde tu última confirmación —sólo los cambios que todavía no están preparados—. Esto puede resultar desconcertante, porque si has preparado todos tus cambios, `git diff` no mostrará nada.

Por poner otro ejemplo, si preparas el archivo `benchmarks.rb` y después lo editas, puedes usar `git diff` para ver las modificaciones del archivo que están preparadas, y las que no lo están:

```
$ git add benchmarks.rb
$ echo '# test line' >> benchmarks.rb
$ git status
# On branch master
#
# Changes to be committed:
#
#   modified:   benchmarks.rb
#
# Changes not staged for commit:
#
#   modified:   benchmarks.rb
#
```

Ahora puedes usar `git diff` para ver qué es lo que aún no está preparado:

```
$ git diff
diff --git a/benchmarks.rb b/benchmarks.rb
index e445e28..86b2f7c 100644
--- a/benchmarks.rb
+++ b/benchmarks.rb
```

```
@@ -127,3 +127,4 @@ end
main()

##pp Grit::GitRuby.cache_client.stats
+# test line
```

Y `git diff --cached` para ver los cambios que llevas preparados hasta ahora:

```
$ git diff --cached
diff --git a/benchmarks.rb b/benchmarks.rb
index 3cb747f..e445e28 100644
--- a/benchmarks.rb
+++ b/benchmarks.rb
@@ -36,6 +36,10 @@ def main
    @commit.parents[0].parents[0].parents[0]
  end

+  run_code(x, 'commits 1') do
+    git.commits.size
+  end
+
  run_code(x, 'commits 2') do
    log = git.commits('master', 15)
    log.size
```

## Confirmando tus cambios

Ahora que el área de preparación está como tú quieres, puedes confirmar los cambios. Recuerda que cualquier cosa que todavía esté sin preparar —cualquier archivo que hayas creado o modificado, y sobre el que no hayas ejecutado `git add` desde su última edición— no se incluirá en esta confirmación. Se mantendrán como modificados en tu disco.

En este caso, la última vez que ejecutaste `git status` viste que estaba todo preparado, por lo que estás listo para confirmar tus cambios. La forma más fácil de confirmar es escribiendo `git commit`:

```
$ git commit
```

Al hacerlo, se ejecutará tu editor de texto. (Esto se configura a través de la variable de entorno `$EDITOR` de tu shell —normalmente vim o emacs, aunque puedes configurarlo usando el comando `git config --global core.editor` como vimos en el Capítulo 1.—)

El editor mostrará el siguiente texto (este ejemplo usa 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:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   README
#       modified:   benchmarks.rb
~
~
~
".git/COMMIT_EDITMSG" 10L, 283C
```

Puedes ver que el mensaje de confirmación predeterminado contiene la salida del comando `git status` comentada, y una línea vacía arriba del todo. Puedes eliminar estos comentarios y escribir tu mensaje de confirmación, o puedes dejarlos para ayudarte a recordar las modificaciones que estás confirmando. (Para un recordatorio todavía más explícito de lo que has modificado, puedes pasar la opción `-v` a `git commit`. Esto provoca que se añadan también las diferencias de tus cambios, para que veas exactamente lo que hiciste.) Cuando sales del editor, Git crea tu confirmación con el mensaje que hayas especificado (omitiendo los comentarios y las diferencias).

Como alternativa, puedes escribir tu mensaje de confirmación desde la propia línea de comandos mediante la opción `-m`:

```
$ git commit -m "Story 182: Fix benchmarks for speed"
[master]: created 463dc4f: "Fix benchmarks for speed"
 2 files changed, 3 insertions(+), 0 deletions(-)
create mode 100644 README
```

¡Acabas de crear tu primera confirmación! Puedes ver que el comando `commit` ha dado cierta información sobre la confirmación: a qué rama has confirmado (master), cuál es su suma de

comprobación SHA-1 de la confirmación (**463dc4f**), cuántos archivos se modificaron, y estadísticas acerca de cuántas líneas se han añadido y cuántas se han eliminado.

Recuerda que la confirmación registra la instantánea de tu área de preparación. Cualquier cosa que no preparases sigue estando modificada; puedes hacer otra confirmación para añadirla a la historia del proyecto. Cada vez que confirmas, estás registrando una instantánea de tu proyecto, a la que puedes volver o con la que puedes comparar más adelante.

## Saltándote el área de preparación

Aunque puede ser extremadamente útil para elaborar confirmaciones exactamente a tu gusto, el área de preparación es en ocasiones demasiado compleja para las necesidades de tu flujo de trabajo. Si quieres saltarte el área de preparación, Git proporciona un atajo. Pasar la opción **-a** al comando **git commit** hace que Git prepare todo archivo que estuviese en seguimiento antes de la confirmación, permitiéndote obviar toda la parte de **git add**:

```
$ git status
# On branch master
#
# Changes not staged for commit:
#
#   modified:   benchmarks.rb
#
$ git commit -a -m 'added new benchmarks'
[master 83e38c7] added new benchmarks
1 files changed, 5 insertions(+), 0 deletions(-)
```

Fíjate que no has tenido que ejecutar **git add** sobre el archivo `benchmarks.rb` antes de hacer la confirmación.

## Eliminando archivos

Para eliminar un archivo de Git, debes eliminarlo de tus archivos bajo seguimiento (más concretamente, debes eliminarlo de tu área de preparación), y después confirmar. El comando **git rm** se encarga de eso, y también elimina el archivo de tu directorio de trabajo, para que no lo veas entre los archivos sin seguimiento.

Si simplemente eliminas el archivo de tu directorio de trabajo, aparecerá bajo la cabecera “Modificados pero no actualizados” (“Changes not staged for commit”) (es decir, *sin preparar*) de la salida del comando **git status**:

```
$ rm grit.gemspec
$ git status
# On branch master
#
# Changes not staged for commit:
#   (use "git add/rm <file>..." to update what will be committed)
#
#       deleted:    grit.gemspec
#
```

Si entonces ejecutas el comando **git rm**, preparas la eliminación del archivo en cuestión:

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

La próxima vez que confirmes, el archivo desaparecerá y dejará de estar bajo seguimiento. Si ya habías modificado el archivo y lo tenías en el área de preparación, deberás forzar su eliminación con la opción **-f**. Ésta es una medida de seguridad para evitar la eliminación accidental de información que no ha sido registrada en una instantánea, y que por tanto no podría ser recuperada.

Otra cosa que puedes hacer es mantener el archivo en tu directorio de trabajo, pero eliminarlo de tu área de preparación. Dicho de otro modo, puedes mantener el archivo en tu disco duro, pero interrumpir su seguimiento por parte de Git. Esto resulta particularmente útil cuando olvidaste añadir algo a tu archivo **.gitignore** y lo añadiste accidentalmente, como un archivo de log enorme, o un montón de archivos **.a**. Para hacer esto, usa la opción **--cached**:

```
$ git rm --cached readme.txt
```

El comando **git rm** acepta archivos, directorios, y patrones glob. Es decir, que podrías hacer algo así:



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

Fíjate en la barra hacia atrás (\) antes del \*. Es necesaria debido a que Git hace su propia expansión de rutas, además de la expansión que hace tu shell. En la consola del sistema de Windows, esta barra debe de ser omitida. Este comando elimina todos los archivos con la extensión .log en el directorio log/. También puedes hacer algo así:

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

Este comando elimina todos los archivos que terminan en ~.

## Moviendo archivos

A diferencia de muchos otros VCSs, Git no hace un seguimiento explícito del movimiento de archivos. Si renombras un archivo, en Git no se almacena ningún metadato que indique que lo has renombrado. Sin embargo, Git es suficientemente inteligente como para darse cuenta —trataremos el tema de la detección de movimiento de archivos un poco más adelante.

Por tanto, es un poco desconcertante que Git tenga un comando mv. Si quieres renombrar un archivo en Git, puedes ejecutar algo así:

```
$ git mv file_from file_to
```

Y funciona perfectamente. De hecho, cuando ejecutas algo así y miras la salida del comando status, verás que Git lo considera un archivo renombrado:

```
$ git mv README.txt README
$ git status
# On branch master
# Your branch is ahead of 'origin/master' by 1 commit.
#
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       renamed:   README.txt -> README
#
```

Sin embargo, esto es equivalente a ejecutar algo así:

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

Git se da cuenta de que es un renombrado de manera implícita, así que no importa si renombas un archivo de este modo, o usando el comando **mv**. La única diferencia real es que **mv** es un comando en vez de tres —es más cómodo—. Y lo que es más importante, puedes usar cualquier herramienta para renombrar un archivo, y preocuparte de los **add** y **rm** más tarde, antes de confirmar.

## 2.3 Fundamentos de Git - Viendo el histórico de confirmaciones

### Viendo el histórico 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 que suelo usar para hacer demostraciones. Para clonar el proyecto, ejecuta:

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

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 code

```
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 argumento, **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 histórico:

```
$ 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.version = "0.1.0"
+  s.version = "0.1.1"
   s.author = "Scott Chacon"

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

removed unnecessary test code

```
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.

A veces es más fácil revisar cambios a nivel de palabra que a nivel de línea. Git dispone de la opción `--word-diff`, que se puede añadir al comando `git log -p` para obtener las diferencias por palabras en lugar de las diferencias línea por línea. Formatear las diferencias a nivel de palabra es bastante inusual cuando se aplica a código fuente, pero resulta muy práctico cuando se aplica a grandes archivos de texto, como libros o tu propia tesis. He aquí un ejemplo:

```
$ git log -U1 --word-diff
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
@@ -7,3 +7,3 @@ spec = Gem::Specification.new do |s|
   s.name    = "simplegit"
```

```
s.version = ["0.1.0-"] {"0.1.1"+}  
s.author  = "Scott Chacon"
```

Como se puede ver, no aparecen líneas añadidas o eliminadas en la salida como en las diferencias normales. Se puede ver la palabra añadida encerrada en `{+ +}` y la eliminada en `[- -]`. Puede que se quiera reducir las usuales tres líneas de contexto en las diferencias a sólo una línea puesto que el contexto es ahora de palabras, no de líneas. Se puede hacer esto con `-U1`, como hicimos en el ejemplo de arriba.

También puedes usar con `git log` una serie de opciones de resumen. Por ejemplo, si quieres 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 files changed, 1 insertions(+), 1 deletions(-)
```

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

```
removed unnecessary test code
```

```
lib/simplegit.rb | 5 -----  
1 files changed, 0 insertions(+), 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(+), 0 deletions(-)

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 code
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 especificas el formato explícitamente, sabes que no cambiará en futuras actualizaciones de Git—:

```
$ git log --pretty=format:"%h - %an, %ar : %s"
ca82a6d - Scott Chacon, 11 months ago : changed the version number
085bb3b - Scott Chacon, 11 months ago : removed unnecessary test code
a11bef0 - Scott Chacon, 11 months ago : first commit
```

La Tabla 2-1 lista algunas de las opciones más útiles aceptadas por **format**.

Opción	Descripción de la salida
%H	Hash de la confirmación
%h	Hash de la confirmación abreviado
%T	Hash del árbol
%t	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 en mayor profundidad en el Capítulo 5.

Las opciones `oneline` y `format` son especialmente útiles combinadas con otra opción llamada `--graph`. Ésta añade un pequeño gráfico ASCII mostrando tu histórico de ramificaciones y uniones, como podemos ver en nuestra copia del repositorio del proyecto Grit:

```
$ 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
```

Éstas son sólo algunas de las opciones para formatear la salida de `git log` —existen muchas más. La Tabla 2-2 lista las opciones vistas hasta ahora, y algunas otras opciones de formateo que pueden resultarte útiles, así como su efecto sobre la salida.

Opción	Descripción
<code>-p</code>	Muestra el parche introducido en cada confirmación.
<code>--word-diff</code>	Muestra el parche en formato de una palabra.
<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).

`--oneline` Un cómodo acortamiento de la opción `--pretty=oneline`  
`--abbrev-commit`.

## Limitando la salida del histórico

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 histórico 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 quieres 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.)

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 la Tabla 2-3 se listan estas opciones, y algunas otras bastante comunes, a modo de referencia.

Opción	Descripción
<code>-(n)</code>	Muestra solamente las últimas <code>n</code> 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.

Por ejemplo, si quieres 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 attribute
acd3b9e - Enhance hold_lock_file_for_{update,append}()
f563754 - demonstrate breakage of detached checkout wi
d1a43f2 - reset --hard/read-tree --reset -u: remove un
51a94af - Fix "checkout --track -b newbranch" on detac
b0ad11e - pull: allow "git pull origin $something:$cur
```

De las casi 20.000 confirmaciones en la historia del código fuente de Git, este comando muestra las 6 que cumplen estas condiciones.

## Usando un interfaz gráfico para visualizar el histórico

Si deseas utilizar una herramienta más gráfica para visualizar el histórico de confirmaciones, puede que quieras echarle un ojo a un programa Tcl/Tk llamado gitk que se distribuye junto con Git. Gitk es básicamente un `git log` visual, y acepta casi todas las opciones de filtrado que acepta `git log`. Si tecleas `gitk` en la línea de comandos dentro de tu proyecto, deberías ver algo como lo de la Figura 2-2.

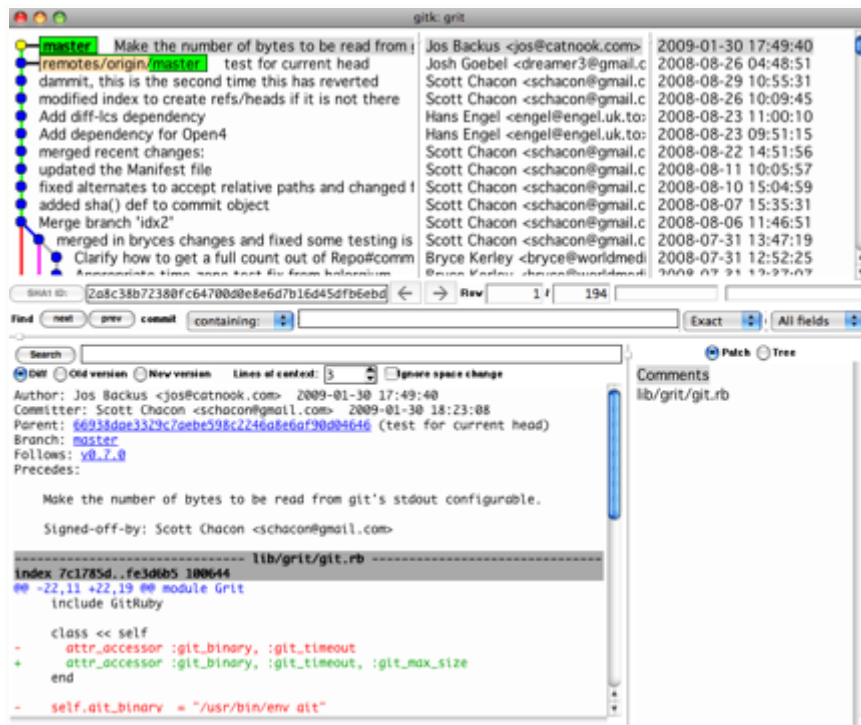


Figura 2-2. El visualizador de histórico gitk.

Puedes ver el histórico de confirmaciones en la mitad superior de la ventana, junto con un gráfico de ascendencia. El visor de diferencias de la mitad inferior muestra las modificaciones introducidas en cada confirmación que selecciones.

## 2.4 Fundamentos de Git - Deshaciendo cosas

### Deshaciendo cosas

En cualquier momento puedes querer deshacer algo. En esta sección veremos algunas herramientas básicas para deshacer cambios. Ten cuidado, porque no siempre puedes volver atrás después de algunas de estas operaciones. Ésta es una de las pocas áreas de Git que pueden provocar que pierdas datos si haces las cosas incorrectamente.

#### [Modificando tu última confirmación](#)

Uno de los casos más comunes en el que quieres deshacer cambios es cuando confirmas demasiado pronto y te olvidas de añadir algún archivo, o te confundes al introducir el mensaje de confirmación. Si quieres volver a hacer la confirmación, puedes ejecutar un `commit` con la opción `--amend`:

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

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

Se lanzará el editor de texto para que introduzcas tu mensaje, pero ya contendrá el mensaje de la confirmación anterior. Puedes editar el mensaje, igual que siempre, pero se sobrescribirá tu confirmación anterior.

Por ejemplo, si confirmas y luego te das cuenta de que se te olvidó preparar los cambios en uno de los archivos que querías añadir, puedes hacer algo así:

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

Estos tres comandos acabarán convirtiéndose en una única confirmación —la segunda confirmación reemplazará los resultados de la primera.

## Deshaciendo la preparación de un archivo

Las dos secciones siguientes muestran cómo trabajar con las modificaciones del área de preparación y del directorio de trabajo. Lo bueno es que el comando que usas para determinar el estado de ambas áreas te recuerda como deshacer sus modificaciones. Por ejemplo, digamos que has modificado dos archivos, y quieres confirmarlos como cambios separados, pero tecleas accidentalmente `git add *` y preparas ambos. ¿Cómo puedes sacar uno de ellos del área de preparación? El comando `git status` te lo recuerda:

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

```
#
#   modified:   README.txt
#   modified:   benchmarks.rb
#
```

Justo debajo de la cabecera “Cambios a confirmar” (“Changes to be committed”), dice que uses `git reset HEAD <archivo>...` para sacar un archivo del área de preparación. Vamos a aplicar ese consejo sobre `benchmarks.rb`:

```
$ git reset HEAD benchmarks.rb
benchmarks.rb: locally modified
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   modified:   README.txt
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#   modified:   benchmarks.rb
#
```

El comando es un poco extraño, pero funciona. El archivo `benchmarks.rb` ahora está modificado, no preparado.

## Deshaciendo la modificación de un archivo

¿Qué pasa si te das cuenta de que no quieres mantener las modificaciones que has hecho sobre el archivo `benchmarks.rb`? ¿Cómo puedes deshacerlas fácilmente —revertir el archivo al mismo estado en el que estaba cuando hiciste tu última confirmación (o cuando clonaste el repositorio, o como quiera que metieses el archivo en tu directorio de trabajo)? Afortunadamente, `git status` también te dice como hacer esto. En la salida del último ejemplo, la cosa estaba 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:  benchmarks.rb
#
```

Te dice de forma bastante explícita cómo descartar las modificaciones que hayas hecho (al menos las versiones de Git a partir de la 1.6.1 lo hacen —si tienes una versión más antigua, te recomendamos encarecidamente que la actualices para obtener algunas de estas mejoras de usabilidad). Vamos a hacer lo que dice:

```
$ git checkout -- benchmarks.rb
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#    modified:   README.txt
#
```

Puedes ver que se han revertido los cambios. También deberías ser consciente del peligro de este comando: cualquier modificación hecha sobre este archivo ha desaparecido —acabas de sobrescribirlo con otro archivo—. Nunca uses este comando a no ser que estés absolutamente seguro de que no quieres el archivo. Si lo único que necesitas es olvidarte de él momentáneamente, veremos los conceptos de apilamiento (stashing) y ramificación (branching) en el próximo capítulo; en general son formas más adecuadas de trabajar.

Recuerda, cualquier cosa que esté confirmada en Git casi siempre puede ser recuperada. Incluso confirmaciones sobre ramas que han sido eliminadas, o confirmaciones sobrescritas con la opción **--amend**, pueden recuperarse (véase el Capítulo 9 para conocer más sobre recuperación de datos). Sin embargo, cualquier cosa que pierdas y que no estuviese confirmada, probablemente no vuelvas a verla nunca más.

## 2.5 Fundamentos de Git - Trabajando con repositorios remotos

### Trabajando con repositorios remotos

Para poder colaborar en cualquier proyecto Git, necesitas saber cómo gestionar tus repositorios remotos. Los repositorios remotos son versiones de tu proyecto que se encuentran alojados en Internet o en algún punto de la red. Puedes tener varios, cada uno de los cuales puede ser de sólo lectura, o de lectura/escritura, según los permisos que tengas. Colaborar con otros implica gestionar estos repositorios remotos, y mandar (push) y recibir (pull) datos de ellos cuando necesites compartir cosas.

Gestionar repositorios remotos implica conocer cómo añadir repositorios nuevos, eliminar aquellos que ya no son válidos, gestionar ramas remotas e indicar si están bajo seguimiento o no, y más cosas. En esta sección veremos todos estos conceptos.

## Mostrando tus repositorios remotos

Para ver qué repositorios remotos tienes configurados, puedes ejecutar el comando **git remote**. Mostrará una lista con los nombres de los remotos que hayas especificado. Si has clonado tu repositorio, deberías ver por lo menos "origin" —es el nombre predeterminado que le da Git al servidor del que clonaste—:

```
$ git clone git://github.com/schacon/ticgit.git
Initialized empty Git repository in /private/tmp/ticgit/.git/
remote: Counting objects: 595, done.
remote: Compressing objects: 100% (269/269), done.
remote: Total 595 (delta 255), reused 589 (delta 253)
Receiving objects: 100% (595/595), 73.31 KiB | 1 KiB/s, done.
Resolving deltas: 100% (255/255), done.
$ cd ticgit
$ git remote
origin
```

También puedes añadir la opción **-v**, que muestra la URL asociada a cada repositorio remoto:

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

Si tienes más de un remoto, este comando los lista todos. Por ejemplo, mi repositorio Grit tiene esta pinta:

```
$ cd grit
$ git remote -v
```



```
bakkdoor git://github.com/bakkdoor/grit.git
cho45    git://github.com/cho45/grit.git
defunkt  git://github.com/defunkt/grit.git
koke     git://github.com/koke/grit.git
origin   git@github.com:mojombo/grit.git
```

Esto significa que podemos recibir contribuciones de cualquiera de estos usuarios de manera bastante fácil. Pero fíjate en que sólo el remoto origen tiene una URL SSH, por lo que es el único al que podemos enviar (veremos el por qué en el Capítulo 4).

## Añadiendo repositorios remotos

Ya he mencionado y he dado ejemplos de repositorios remotos en secciones anteriores, pero a continuación veremos cómo añadirlos explícitamente. Para añadir un nuevo repositorio Git remoto, asignándole un nombre con el que referenciarlo fácilmente, ejecuta `git remote add [nombre] [url]`:

```
$ git remote
origin
$ git remote add pb git://github.com/paulboone/ticgit.git
$ git remote -v
origin git://github.com/schacon/ticgit.git
pb git://github.com/paulboone/ticgit.git
```

Ahora puedes usar la cadena "pb" en la línea de comandos, en lugar de toda la URL. Por ejemplo, si quieres recuperar toda la información de Paul que todavía no tienes en tu repositorio, puedes ejecutar `git fetch pb`:

```
$ git fetch pb
remote: Counting objects: 58, done.
remote: Compressing objects: 100% (41/41), done.
remote: Total 44 (delta 24), reused 1 (delta 0)
Unpacking objects: 100% (44/44), done.
From git://github.com/paulboone/ticgit
* [new branch]    master    -> pb/master
* [new branch]    ticgit    -> pb/ticgit
```

La rama maestra de Paul es accesible localmente como `pb/master` —puedes unirla a una de tus ramas, o copiarla localmente para inspeccionarla.

## Recibiendo de tus repositorios remotos

Como acabas de ver, para recuperar datos de tus repositorios remotos puedes ejecutar:

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

Este comando recupera todos los datos del proyecto remoto que no tengas todavía. Después de hacer esto, deberías tener referencias a todas las ramas del repositorio remoto, que puedes unir o inspeccionar en cualquier momento. (Veremos qué son las ramas y cómo utilizarlas en más detalle en el Capítulo 3.)

Si clonas un repositorio, el comando añade automáticamente ese repositorio remoto con el nombre de "origin". Por tanto, `git fetch origin` recupera toda la información enviada a ese servidor desde que lo clonaste (o desde la última vez que ejecutaste `fetch`). Es importante tener en cuenta que el comando `fetch` sólo recupera la información y la pone en tu repositorio local —no la une automáticamente con tu trabajo ni modifica aquello en lo que estás trabajando. Tendrás que unir ambos manualmente a posteriori.

Si has configurado una rama para seguir otra rama remota (véase la siguiente sección y el Capítulo 3 para más información), puedes usar el comando `git pull` para recuperar y unir automáticamente la rama remota con tu rama actual. Éste puede resultarte un flujo de trabajo más sencillo y más cómodo; y por defecto, el comando `git clone` automáticamente configura tu rama local maestra para que siga la rama remota maestra del servidor del cual clonaste (asumiendo que el repositorio remoto tiene una rama maestra). Al ejecutar `git pull`, por lo general se recupera la información del servidor del que clonaste, y automáticamente se intenta unir con el código con el que estás trabajando actualmente.

## Enviando a tus repositorios remotos

Cuando tu proyecto se encuentra en un estado que quieres compartir, tienes que enviarlo a un repositorio remoto. El comando que te permite hacer esto es sencillo: `git push [nombre-remoto][nombre-rama]`. Si quieres enviar tu rama maestra (`master`) a tu servidor origen (`origin`), ejecutarías esto para enviar tu trabajo al servidor:

```
$ git push origin master
```

Este comando funciona únicamente si has clonado de un servidor en el que tienes permiso de escritura, y nadie ha enviado información mientras tanto. Si tú y otra persona clonáis a la vez, y él envía su información y luego envías tú la tuya, tu envío será rechazado. Tendrás que bajarte

primero su trabajo e incorporarlo en el tuyo para que se te permita hacer un envío. Véase el Capítulo 3 para ver en detalle cómo enviar a servidores remotos.

## Inspeccionando un repositorio remoto

Si quieres ver más información acerca de un repositorio remoto en particular, puedes usar el comando `git remote show [nombre]`. Si ejecutas este comando pasándole el nombre de un repositorio, como `origin`, obtienes algo así:

```
$ git remote show origin
* remote origin
URL: git://github.com/schacon/ticgit.git
Remote branch merged with 'git pull' while on branch master
  master
Tracked remote branches
  master
  ticgit
```

Esto lista la URL del repositorio remoto, así como información sobre las ramas bajo seguimiento. Este comando te recuerda que si estás en la rama maestra y ejecutas `git pull`, automáticamente unirá los cambios a la rama maestra del remoto después de haber recuperado todas las referencias remotas. También lista todas las referencias remotas que ha recibido.

El anterior es un sencillo ejemplo que te encontrarás con frecuencia. Sin embargo, cuando uses Git de forma más avanzada, puede que `git remote show` muestre mucha más información:

```
$ git remote show origin
* remote origin
URL: git@github.com:defunkt/github.git
Remote branch merged with 'git pull' while on branch issues
  issues
Remote branch merged with 'git pull' while on branch master
  master
New remote branches (next fetch will store in remotes/origin)
  caching
Stale tracking branches (use 'git remote prune')
  libwalker
  walker2
Tracked remote branches
  acl
  apiv2
```

```
dashboard2
issues
master
postgres
Local branch pushed with 'git push'
master:master
```

Este comando muestra qué rama se envía automáticamente cuando ejecutas `git push` en determinadas ramas. También te muestra qué ramas remotas no tienes todavía, qué ramas remotas tienes y han sido eliminadas del servidor, y múltiples ramas que serán unidas automáticamente cuando ejecutes `git pull`.

### Eliminando y renombrando repositorios remotos

Si quieres renombrar una referencia a un repositorio remoto, en versiones recientes de Git puedes ejecutar `git remote rename`. Por ejemplo, si quieres renombrar `pb` a `paul`, puedes hacerlo de la siguiente manera:

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

Conviene mencionar que esto también cambia el nombre de tus ramas remotas. Lo que antes era referenciado en `pb/master` ahora está en `paul/master`.

Si por algún motivo quieres eliminar una referencia —has movido el servidor o ya no estás usando un determinado mirror, o quizás un contribuidor ha dejado de contribuir— puedes usar el comando `git remote rm`:

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

## 2.6 Fundamentos de Git - Creando etiquetas

## Creando etiquetas

Como muchos VCSs, Git tiene la habilidad de etiquetar (tag) puntos específicos en la historia como importantes. Generalmente la gente usa esta funcionalidad para marcar puntos donde se ha lanzado alguna versión (v1.0, y así sucesivamente). En esta sección aprenderás cómo listar las etiquetas disponibles, crear nuevas etiquetas y qué tipos diferentes de etiquetas hay.

### Listando 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 es realmente importante.

También puedes buscar etiquetas de acuerdo a un patrón en particular. El repositorio fuente de Git, por ejemplo, contiene más de 240 etiquetas. Si solo estás interesado en la serie 1.4.2, puedes ejecutar esto:

```
$ git tag -l 'v1.4.2.*'
v1.4.2.1
v1.4.2.2
v1.4.2.3
v1.4.2.4
```

### Creando etiquetas

Git usa dos tipos principales de etiquetas: ligeras y anotadas. Una etiqueta ligera es muy parecida a una rama que no cambia —un puntero a una confirmación específica—. Sin embargo, las etiquetas anotadas son almacenadas como objetos completos en la base de datos de Git. Tienen suma de comprobación; contienen el nombre del etiquetador, correo electrónico y fecha; tienen mensaje de etiquetado; y pueden estar firmadas y verificadas con GNU Privacy Guard (GPG). Generalmente se recomienda crear etiquetas anotadas para disponer de toda esta información; pero si por alguna razón quieres una etiqueta temporal y no quieres almacenar el resto de información, también tiene disponibles las etiquetas ligeras.

## Etiquetas anotadas

Crear una etiqueta anotada en Git es simple. La forma más fácil es especificar **-a** al ejecutar el comando **tag**:

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

El parámetro **-m** especifica el mensaje, el cual se almacena con la etiqueta. Si no se especifica un mensaje para la etiqueta anotada, Git lanza tu editor para poder escribirlo.

Puedes ver los datos de la etiqueta junto con la confirmación que fue etiquetada usando el comando **git show**:

```
$ git show v1.4
tag v1.4
Tagger: Scott Chacon <schacon@gee-mail.com>
Date: Mon Feb 9 14:45:11 2009 -0800

my version 1.4
commit 15027957951b64cf874c3557a0f3547bd83b3ff6
Merge: 4a447f7... a6b4c97...
Author: Scott Chacon <schacon@gee-mail.com>
Date: Sun Feb 8 19:02:46 2009 -0800

Merge branch 'experiment'
```

Esto muestra la información del autor de la etiqueta, la fecha en la que la confirmación fue etiquetada, y el mensaje de anotación antes de mostrar la información de la confirmación.

## Etiquetas firmadas

También puedes firmar tus etiquetas con GPG, siempre que tengas una clave privada. Lo único que debes hacer es usar **-s** en vez de **-a**:

```
$ 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@gee-mail.com>"
```

1024-bit DSA key, ID F721C45A, created 2009-02-09

Si ejecutas `git show` en esa etiqueta, puedes ver la firma GPG adjunta a ella:

```
$ git show v1.5
tag v1.5
Tagger: Scott Chacon <schacon@gee-mail.com>
Date: Mon Feb 9 15:22:20 2009 -0800
```

```
my signed 1.5 tag
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1.4.8 (Darwin)
```

```
iEYEABECAAYFAkmQurIACgkQON3DxfchxFr5cACeIMN+ZxLKggJQf0QYiQBwgySN
Ki0An2JeAVUCAiJ7Ox6ZEtK+NvZAJ82/
=WryJ
-----END PGP SIGNATURE-----
commit 15027957951b64cf874c3557a0f3547bd83b3ff6
Merge: 4a447f7... a6b4c97...
Author: Scott Chacon <schacon@gee-mail.com>
Date: Sun Feb 8 19:02:46 2009 -0800
```

Merge branch 'experiment'

Más tarde, aprenderás cómo verificar etiquetas firmadas.

## Etiquetas ligeras

Otra forma de etiquetar confirmaciones es con una etiqueta ligera. Esto es básicamente la suma de comprobación de la confirmación almacenada en un archivo —ninguna otra información es guardada—. Para crear una etiqueta ligera no añadas las opciones `-a`, `-s` o `-m`:

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

Esta vez, si ejecutas el comando **git show** en la etiqueta, no verás ninguna información extra. El comando simplemente muestra la confirmación.

```
$ git show v1.4-lw
commit 15027957951b64cf874c3557a0f3547bd83b3ff6
Merge: 4a447f7... a6b4c97...
Author: Scott Chacon <schacon@gee-mail.com>
Date: Sun Feb 8 19:02:46 2009 -0800
```

Merge branch 'experiment'

## Verificando etiquetas

Para verificar una etiqueta firmada, debes usar **git tag -v [tag-name]**. Este comando utiliza GPG para verificar la firma. Necesitas la clave pública del autor de la firma en tu llavero para que funcione correctamente.

```
$ 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 clave pública del autor de la firma, se obtiene algo parecido a:

```
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'
```

## Etiquetando más tarde



Puedes incluso etiquetar confirmaciones después de avanzar sobre ellas. Supón que tu historico de confirmaciones se parece a esto:

```
$ git log --pretty=oneline
15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'
a6b4c97498bd301d84096da251c98a07c7723e65 beginning write support
0d52aaab4479697da7686c15f77a3d64d9165190 one more thing
6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'
0b7434d86859cc7b8c3d5e1dddfed66ff742fcbc added a commit function
4682c3261057305bdd616e23b64b0857d832627b added a todo file
166ae0c4d3f420721acbb115cc33848dfcc2121a started write support
9fceb02d0ae598e95dc970b74767f19372d61af8 updated rakefile
964f16d36dfccde844893cac5b347e7b3d44abbc commit the todo
8a5cbc430f1a9c3d00faaeffd07798508422908a updated readme
```

Ahora, supón que olvidaste etiquetar el proyecto en v1.2, que estaba en la confirmación "updated rakefile". Puedes hacerlo ahora. Para etiquetar esa confirmación especifica la suma de comprobación de la confirmación (o una parte de la misma) al final del comando:

```
$ git tag -a v1.2 -m 'version 1.2' 9fceb02
```

Puedes ver que has etiquetado la confirmación:

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

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

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

updated rakefile

...

## Compartiendo etiquetas

Por defecto, el comando `git push` no transfiere etiquetas a servidores remotos. Tienes que enviarlas explícitamente a un servidor compartido después de haberlas creado. Este proceso es igual a compartir ramas remotas —puedes ejecutar `git push origin [tagname]`.

```
$ git push origin v1.5
Counting objects: 50, done.
Compressing objects: 100% (38/38), done.
Writing objects: 100% (44/44), 4.56 KiB, done.
Total 44 (delta 18), reused 8 (delta 1)
To git@github.com:schacon/simplegit.git
* [new tag]      v1.5 -> v1.5
```

Si tienes un montón de etiquetas que quieres enviar a la vez, también puedes usar la opción `--tags` en el comando `git push`. Esto transfiere todas tus etiquetas que no estén ya en el servidor remoto.

```
$ git push origin --tags
Counting objects: 50, done.
Compressing objects: 100% (38/38), done.
Writing objects: 100% (44/44), 4.56 KiB, done.
Total 44 (delta 18), reused 8 (delta 1)
To git@github.com:schacon/simplegit.git
* [new tag]      v0.1 -> v0.1
* [new tag]      v1.2 -> v1.2
* [new tag]      v1.4 -> v1.4
* [new tag]      v1.4-lw -> v1.4-lw
* [new tag]      v1.5 -> v1.5
```

Ahora, cuando alguien clone o reciba de tu repositorio, obtendrá también todas tus etiquetas.

## 2.7 Fundamentos de Git - Consejos y trucos

### Consejos y trucos

Antes de que terminemos este capítulo de Git básico, unos pocos trucos y consejos que harán de tu experiencia con Git más sencilla, fácil, o más familiar. Mucha gente usa Git sin usar ninguno de estos consejos, y no nos referiremos a ellos o asumiremos que los has usado más tarde en el libro, pero probablemente debas saber cómo hacerlos.

#### Autocompletado

Si usas el shell Bash, Git viene con un buen script de autocompletado que puedes activar.

Descárgalo directamente desde el código fuente de Git en

<https://github.com/git/git/blob/master/contrib/completion/git-completion.bash>, copia este fichero en tu directorio `home` y añade esto a tu archivo `.bashrc`:

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

Si quieres que Git tenga automáticamente autocompletado para todos los usuarios, copia este script en el directorio `/opt/local/etc/bash_completion.d` en sistemas Mac, o en el directorio `/etc/bash_completion.d/` en sistemas Linux. Este es un directorio de scripts que Bash cargará automáticamente para proveer de autocompletado.

Si estás usando Windows con el Bash de Git, el cual es el predeterminado cuando instalas Git en Windows con `msysGit`, el autocompletado debería estar preconfigurado.

Presiona el tabulador cuando estés escribiendo un comando de Git, y deberían aparecer un conjunto de sugerencias para que escojas:

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

En este caso, escribiendo `git co` y presionando el tabulador dos veces sugiere `commit` y `config`. Añadiendo `m` y pulsando el tabulador completa `git commit` automáticamente.

Esto también funciona con opciones, que probablemente es más útil. Por ejemplo, si quieres ejecutar **git log** y no recuerdas una de las opciones, puedes empezar a escribirla y presionar el tabulador para ver qué coincide:

```
$ git log --s<tab>
--shortstat --since= --src-prefix= --stat --summary
```

Es un pequeño truco que puede guardarte algún tiempo y lectura de documentación.

## Alias de Git

Git no infiere tu comando si lo escribes parcialmente. Si no quieres escribir el texto entero de cada uno de los comandos de Git, puedes establecer fácilmente un alias para cada comando usando **git config**. Aquí hay un par de ejemplos que tal vez quieras establecer:

```
$ 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 vez de escribir **git commit**, simplemente necesitas escribir **git ci**. A medida que uses Git, probablemente uses otros comandos de forma frecuente. En este caso no dudes en crear nuevos alias.

Esta técnica también puede ser muy útil para crear comandos que creas que deben existir. Por ejemplo, para corregir el problema de usabilidad que encontramos al quitar del área de preparación un archivo, puedes añadir tu propio alias:

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

Esto hace los siguientes dos comandos equivalentes:

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

Esto parece un poco mas claro. También es común añadir un comando **last**, tal que así:

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

De esta forma puedes ver la última confirmación fácilmente:

```
$ 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 reemplaza el nuevo comando con lo que le pongas como alias. Sin embargo, tal vez quieres ejecutar un comando externo en lugar de un subcomando de Git. En este caso, empieza el comando con el caracter **!**. Esto es útil si escribes tus propias herramientas que trabajan con un repositorio de Git. Podemos demostrarlo creando el alias **git visual** para ejecutar **gitk**:

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

# Chapter 3

## Ramificaciones en Git

Cualquier sistema de control de versiones moderno tiene algún mecanismo para soportar distintos ramales. Cuando hablamos de ramificaciones, significa que tu 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. ¿Porqué 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 tremendamente 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.

### 3.1 Ramificaciones en Git - ¿Qué es una rama?

#### ¿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 el capítulo 1, Git no los almacena de forma incremental (guardando solo diferencias), sino que los almacena como una serie de instantáneas (copias puntuales de los archivos completos, tal y como se encuentran en ese momento).

En cada confirmación de cambios (commit), Git almacena un punto de control que conserva: un apuntador a la copia puntual de los contenidos preparados (staged), 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 mas 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 el capítulo 1), almacena una copia de cada uno en el repositorio (estas copias se denominan "blobs"), y guarda cada suma de control en el área de preparación (staging area):

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

Cuando creas una confirmación con el comando **git commit**, Git realiza sumas de control de cada subcarpeta (en el ejemplo, solamente tenemos la carpeta principal del proyecto), y guarda en el repositorio Git una copia de cada uno de los archivos contenidos en ella/s. Después, Git crea un objeto de confirmación con los metadatos pertinentes y un apuntador al nodo correspondiente del árbol de proyecto. Esto permitirá poder regenerar posteriormente dicha instantánea cuando sea necesario.

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 de la carpeta (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. Conceptualmente, el contenido del repositorio Git será algo parecido a la Figura 3-1

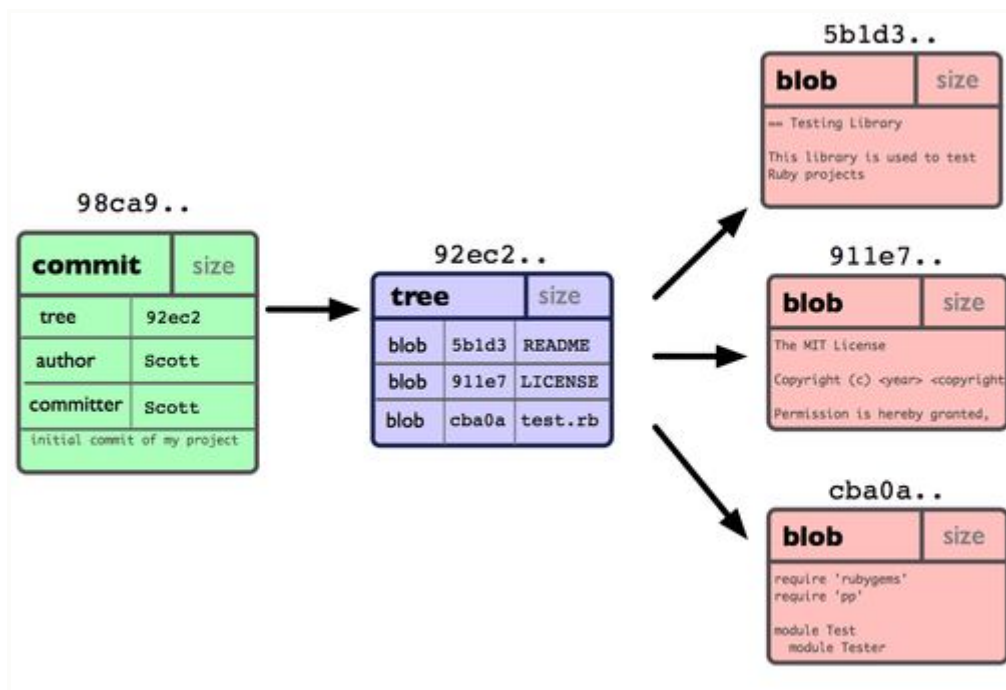


Figura 3-1. Datos en el repositorio tras una confirmación sencilla.

Si haces más cambios y vuelves a confirmar, la siguiente confirmación guardará un apuntador a esta su confirmación precedente. Tras un par de confirmaciones más, el registro ha de ser algo parecido a la Figura 3-2.

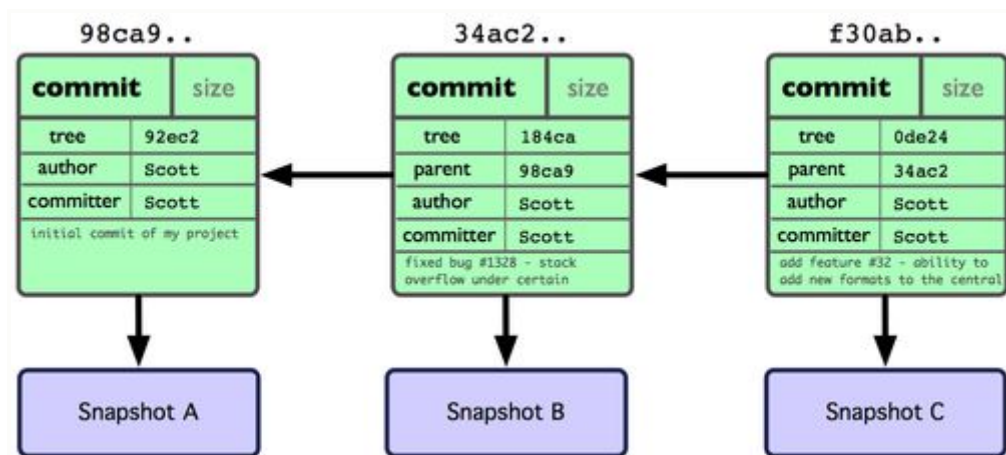


Figura 3-2. Datos en el repositorio tras una serie de confirmaciones.

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. Y la rama **master** apuntará siempre a la última confirmación realizada.

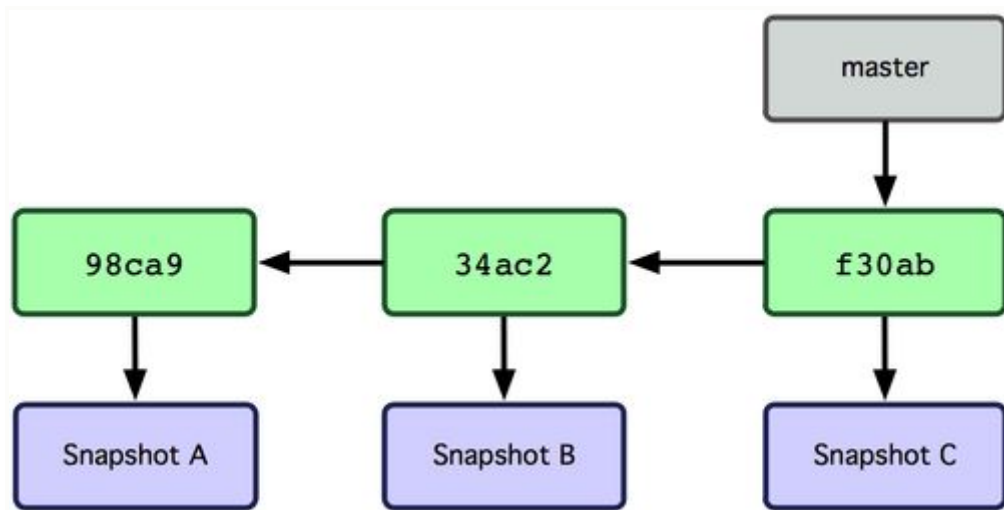


Figura 3-3. Apuntadores en el registro de confirmaciones de una rama.

¿Qué sucede cuando creas una nueva rama? Bueno..., simplemente se crea un nuevo apuntador para que lo puedas mover libremente. Por ejemplo, si quieres crear una nueva rama denominada "testing". Usarás el comando **git branch**:

```
$ git branch testing
```

Esto creará un nuevo apuntador apuntando a la misma confirmación donde estés actualmente (ver Figura 3-4).

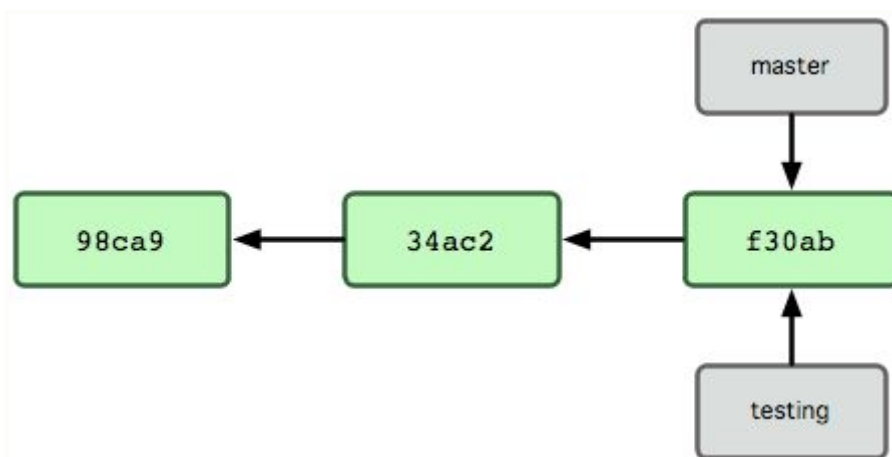


Figura 3-4. Apuntadores de varias ramas en el registro de confirmaciones de cambio.

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, en la rama **master**. Puesto que el comando `git branch` solamente crea una nueva rama, y no salta a dicha rama.

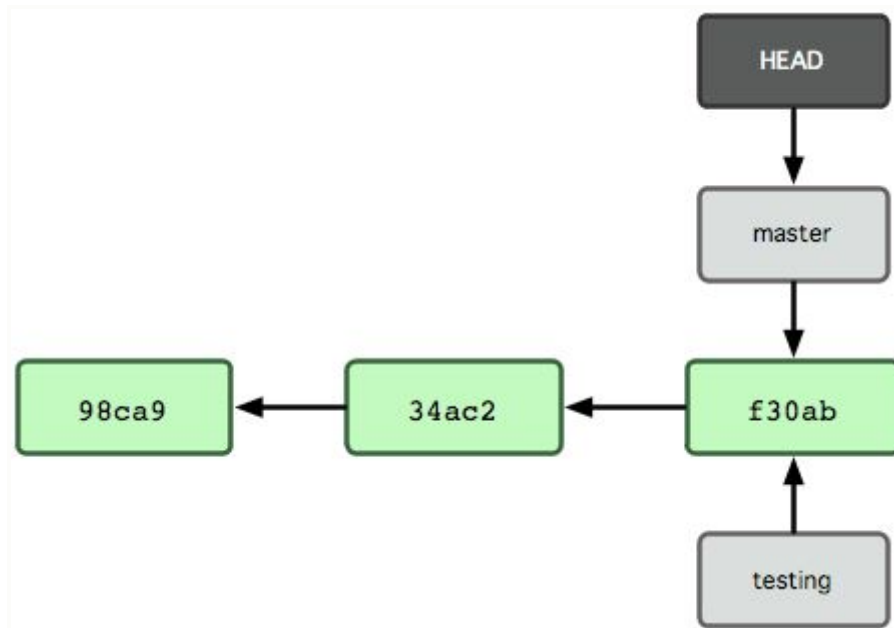


Figura 3-5. Apuntador HEAD a la rama donde estás actualmente.

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** (ver Figura 3-6).

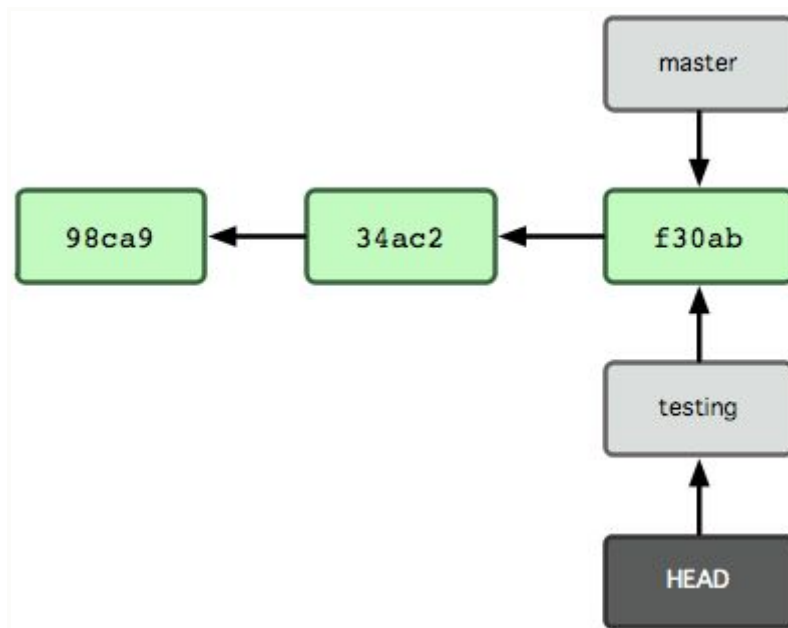


Figura 3-6. Apuntador HEAD apuntando a otra rama cuando saltamos de rama.

¿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'
```

La Figura 3-7 ilustra el resultado.

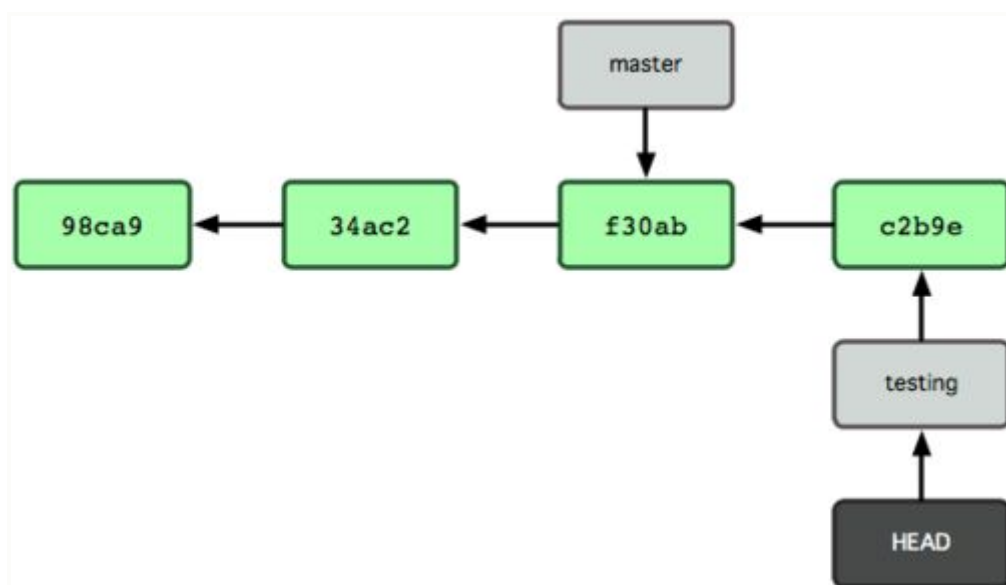


Figura 3-7. La rama apuntada por HEAD avanza con cada confirmación de cambios.

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

Volvamos ahora a la rama **master**:

```
$ git checkout master
```

La Figura 3-8 muestra el resultado.

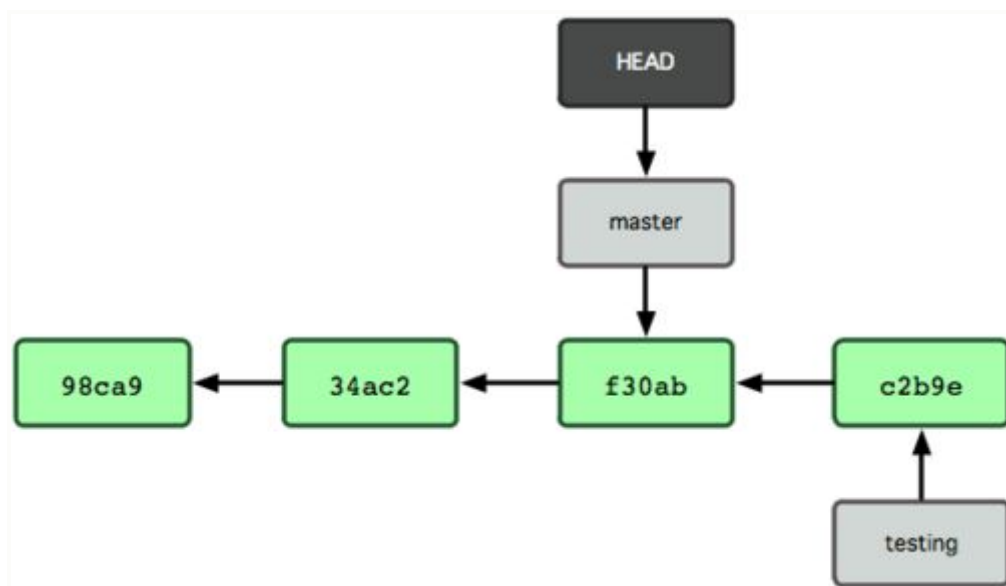


Figura 3-8. HEAD apunta a otra rama cuando hacemos un checkout.

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.

Haz algunos cambios más y confírmalos:

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

Ahora el registro de tu proyecto diverge (ver Figura 3-9). Has creado una rama y saltado a ella, has trabajado sobre ella; has vuelto a la rama original, y has trabajado también sobre ella. Los cambios realizados en ambas sesiones de trabajo están aislados en ramas independientes: puedes saltar libremente de una a otra según estimes oportuno. Y todo ello simplemente con dos comandos: `git branch` y `git checkout`.

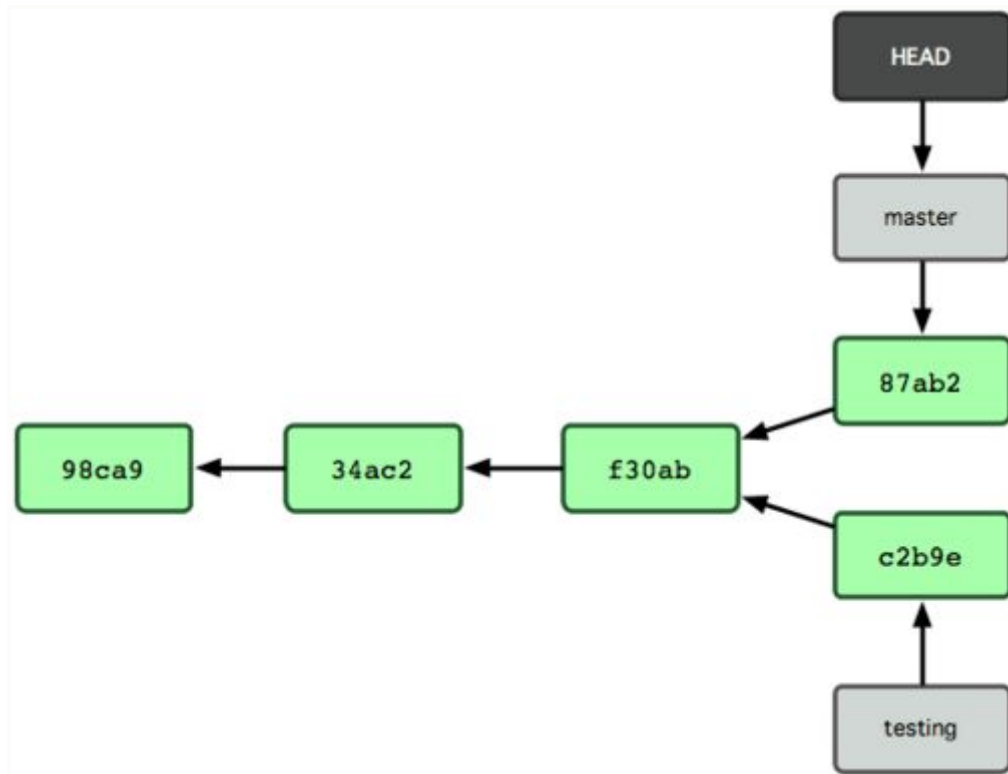


Figura 3-9. Los registros de las ramas divergen.

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 nueva rama supone el copiar todos los archivos del proyecto a una nueva carpeta adicional. Lo que 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.

Y vamos a ver el por qué merece la pena hacerlo así.

## 3.2 Ramificaciones en Git - Procedimientos básicos para ramificar y fusionar

### **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 continuas tu trabajo.

### **Procedimientos básicos de ramificación**

Imagina que estas trabajando en un proyecto, y tienes un par de confirmaciones (commit) ya realizadas. (ver Figura 3-10)

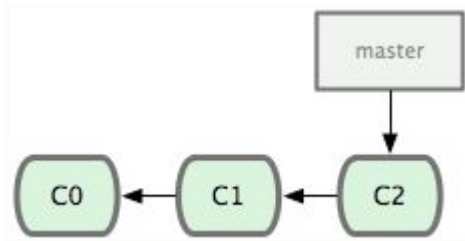


Figura 3-10. Un registro de confirmaciones simple y corto.

Decides trabajar el problema #53, del sistema que tu compañía utiliza para llevar seguimiento de los problemas. Aunque, por supuesto, Git no está ligado a ningún sistema de seguimiento de problemas concreto. Como el problema #53 es un tema concreto y puntual en el que vas a trabajar, creas una nueva rama para él. Para crear una nueva rama y saltar a ella, en un solo paso, puedes utilizar el comando `git checkout` con la opción `-b`:

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

Esto es un atajo a:

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

Figura 3-11 muestra el resultado.

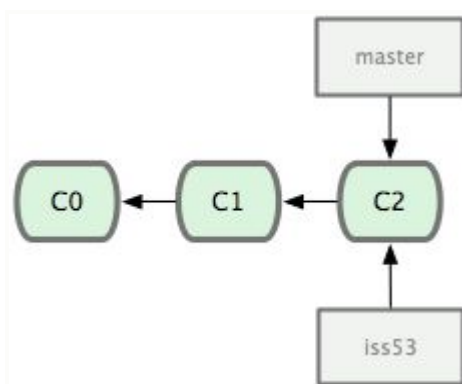


Figura 3-11. Creación de un apuntador a la nueva rama.

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; ver Figura 3-12):

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

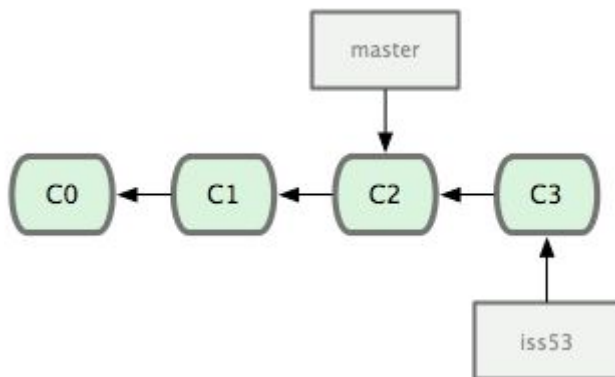


Figura 3-12. La rama **iss53** ha avanzado con tu trabajo.

Entonces, recibes una llamada avisándote de otro problema urgente en el sitio web. Problema que has de resolver inmediatamente. Usando 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 ella.

Pero, antes de poder hacer eso, hemos de tener en cuenta que teniendo cambios aún no confirmados en la carpeta 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 commit amend), que vamos a ver más adelante. 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 la carpeta 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 la carpeta 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 asegurarte que tu copia de trabajo es exactamente tal y como era la rama en la última confirmación de cambios realizada sobre ella.

Volviendo al problema urgente. Vamos a crear una nueva rama **hotfix**, sobre la que trabajar hasta resolverlo (ver Figura 3-13):

```
$ git checkout -b 'hotfix'
Switched to a new branch "hotfix"
$ vim index.html
$ git commit -a -m 'fixed the broken email address'
[hotfix]: created 3a0874c: "fixed the broken email address"
1 files changed, 0 insertions(+), 1 deletions(-)
```

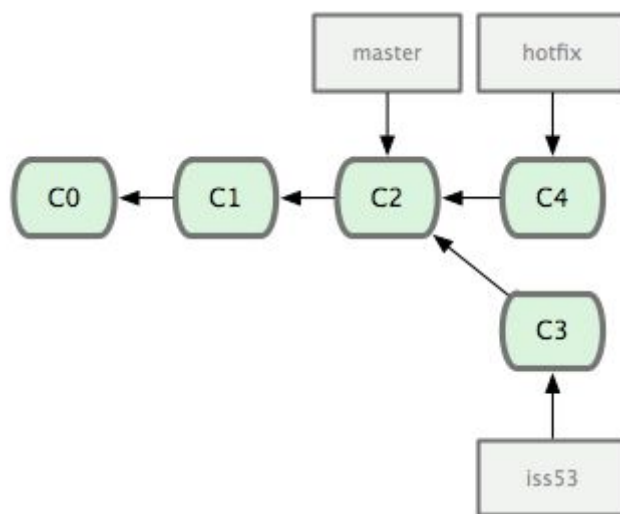


Figura 3-13. rama **hotfix** basada en la rama **master** original.

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

```
$ git checkout master
$ git merge hotfix
Updating f42c576..3a0874c
Fast forward
 README | 1 -
1 files changed, 0 insertions(+), 1 deletions(-)
```

Merece destacar la frase "Avance rápido" ("Fast forward") que aparece en la respuesta al comando. Git ha movido el apuntador hacia adelante, ya que la confirmación apuntada en la rama donde has fusionado estaba directamente "aguas arriba" respecto de la confirmación actual. Dicho de otro modo: cuando intentas fusionar una confirmación con otra confirmación accesible siguiendo directamente el registro 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 (ver Figura 3-14)

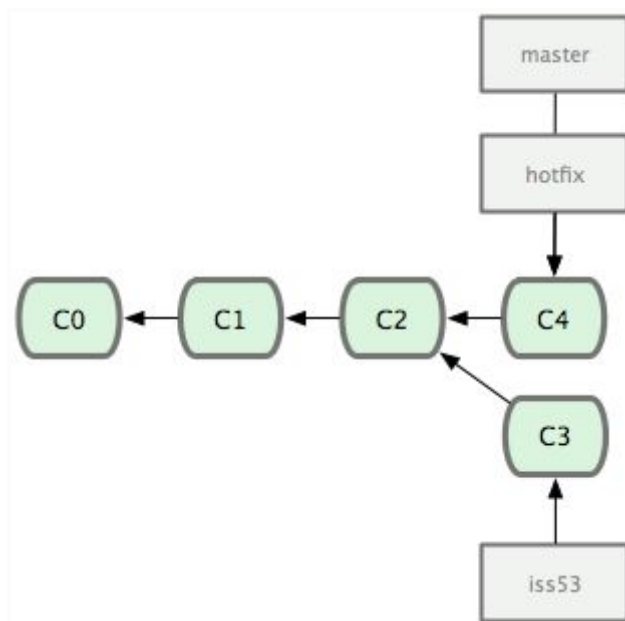


Figura 3-14. Tras la fusión (merge), la rama **master** apunta al mismo sitio que la rama **hotfix**.

Tras haber resuelto el problema urgente que te habí interrumpido tu trabajo, puedes volver a donde estabas. Pero antes, es interesante 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 dispuesto para regresar al trabajo sobre el problema #53 (ver Figura 3-15):

```

$ git checkout iss53
Switched to branch "iss53"
$ vim index.html
$ git commit -a -m 'finished the new footer [issue 53]'
[iss53]: created ad82d7a: "finished the new footer [issue 53]"
1 files changed, 1 insertions(+), 0 deletions(-)

```

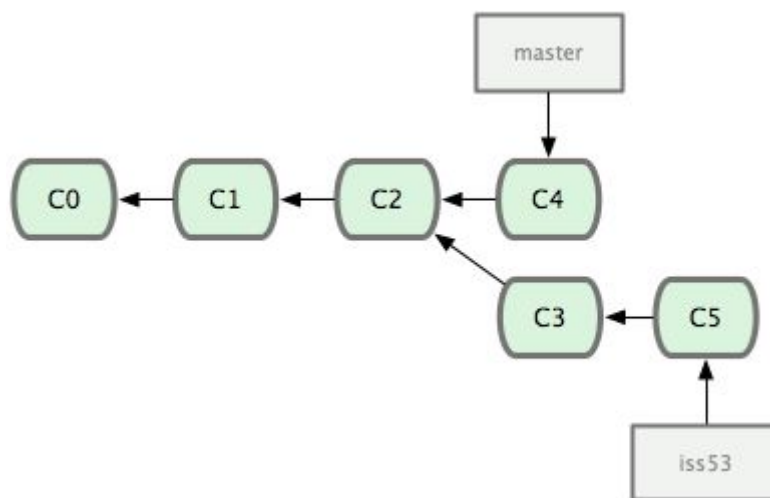


Figura 3-15. La rama **iss53** puede avanzar independientemente.

Cabe indicar 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 llevar (pull) la rama **iss53** a la rama **master**.

## Procedimientos básicos de fusión

Supongamos que tu trabajo con el problema #53 está ya 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, activando (checkout) la rama donde deseas fusionar y lanzando el comando **git merge**:

```

$ git checkout master
$ git merge iss53
Merge made by recursive.
README | 1 +
1 files changed, 1 insertions(+), 0 deletions(-)

```

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 dos. La figura 3-16 ilustra las tres instantáneas que Git utiliza para realizar la fusión en este caso.

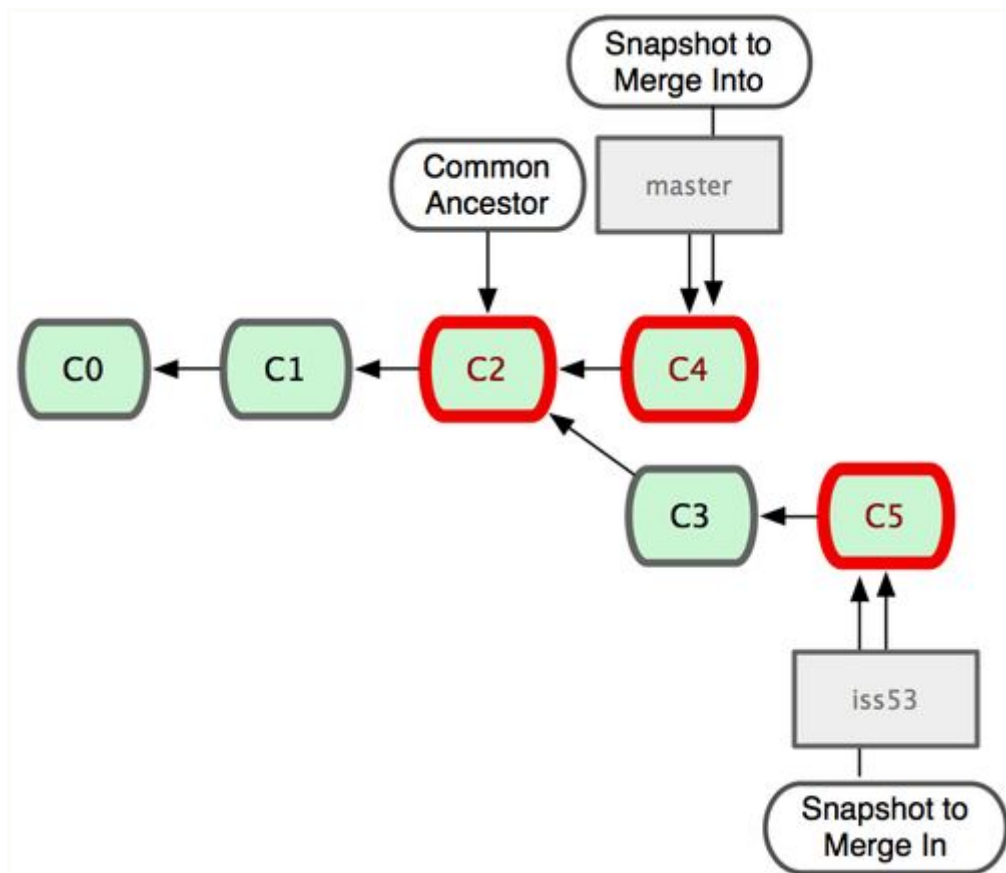


Figura 3-16. Git identifica automáticamente el mejor ancestro común para realizar la fusión de las ramas.

En lugar de simplemente avanzar el apuntador de la rama, Git crea una nueva instantánea (snapshot) resultante de la fusión a tres bandas; y crea automáticamente una nueva confirmación de cambios (commit) que apunta a ella. Nos referimos a este proceso como "fusión confirmada". Y se diferencia en que tiene más de un padre.

Merece la pena destacar el hecho de que es el propio Git quien determina automáticamente el mejor ancestro común para realizar la fusión. Diferenciándose de otros sistemas tales como CVS

o Subversion, donde es el desarrollador quien ha de imaginarse cuál puede ser dicho mejor ancestro común. Esto hace que en Git sea mucho más fácil el realizar fusiones.

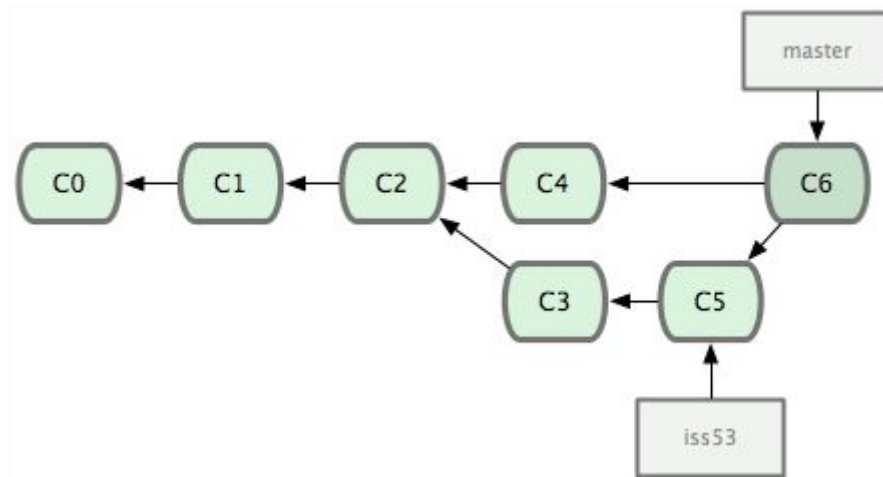


Figura 3-17. Git crea automáticamente una nueva confirmación para la fusión.

Ahora que todo tu trabajo está ya fusionado con la rama principal, ya 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**. Puedes obtener un conflicto de fusión tal que:

```
$ 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 tu resuelvas el conflicto. Para ver qué archivos permanecen

sin fusionar en un determinado momento conflictivo de una fusión, puedes usar el comando **git status**:

```
[master*]$ git status
index.html: needs merge
# 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)
#
#   unmerged:   index.html
#
```

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. Marcadores 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 contenido de uno o de otro lado. Por ejemplo, puedes optar por cambiar el bloque, dejándolo tal que:

```
<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 (staging), 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**. Esto arrancará la correspondiente herramienta de visualización y te permitirá ir resolviendo conflictos con ella.

```
$ git mergetool
merge tool candidates: kdiff3 tkdiff xxdiff meld gvimdiff opendiff emerge vimdiff
Merging the files: index.html
```

Normal merge conflict for 'index.html':

{local}: modified

{remote}: modified

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 un Mac), puedes escogerla entre la lista de herramientas soportadas mostradas al principio ("merge tool candidates"). Tecleando el nombre de dicha herramienta. En el capítulo 7 se verá cómo cambiar este valor por defecto de tu entorno de trabajo.

Tras salir de la herramienta de fusión, Git preguntará a ver 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
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   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.
#
```

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 porqué has hecho lo que has hecho; a no ser que resulte obvio, claro está.

## 3.3 Ramificaciones en Git - Gestión de ramificaciones

### Gestión de ramificaciones

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 profusamente.

El comando `git branch` tiene más funciones que las de crear y borrar ramas. Si lo lanzas sin argumentos, 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. 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, desde la versión 1.5.6, las opciones `--merged` y `--no-merged`. Si deseas ver las ramas que han sido fusionadas en la rama activa, puedes lanzar el comando `git branch --merged`:

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

Aparece la rama `iss53` porque ya ha sido fusionada. Y no lleva por delante el carácter `*` porque todo su contenido ya ha sido incorporado a otras ramas. Podemos borrarla tranquilamente con `git branch -d`, sin miedo a perder nada.

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

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

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

```
$ git branch -d testing
error: The branch 'testing' is not an ancestor of your current HEAD.
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 lo indica el mensaje de ayuda.

## 3.4 Ramificaciones en Git - Flujos de trabajo ramificados

## 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 de una rama a otra multitud de 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 frecuentes fusiones 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). Teniendo otras ramas paralelas denominadas **desarrollo** o **siguiente**, en las que trabajan y realizan pruebas. Estas ramas paralelas no suele estar siempre en un estado estable; pero cada vez que sí lo están, pueden ser fusionadas con la rama **master**. También es habitual el incorporar (pull) ramas puntuales (ramas temporales, como la rama **iss53** del anterior ejemplo) 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 registro de confirmaciones. Mientras que las ramas avanzadas, las que van abriendo camino, apuntan hacia posiciones más recientes.

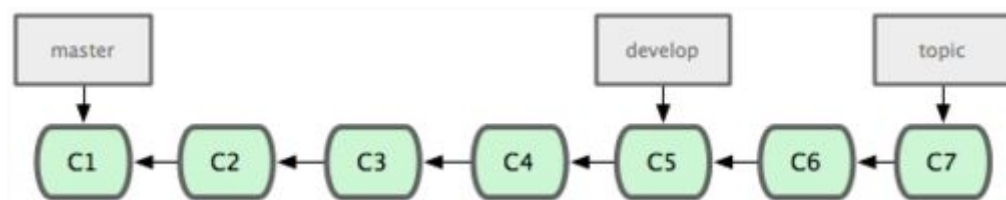


Figura 3-18. Las ramas más estables apuntan hacia posiciones más antiguas en el registro de cambios.

Podría ser más sencillo pensar en las ramas como si fueran silos de almacenamiento. Donde grupos de confirmaciones de cambio (commits) van promocionando hacia silos más estables a medida que son probados y depurados (ver Figura 3-19)

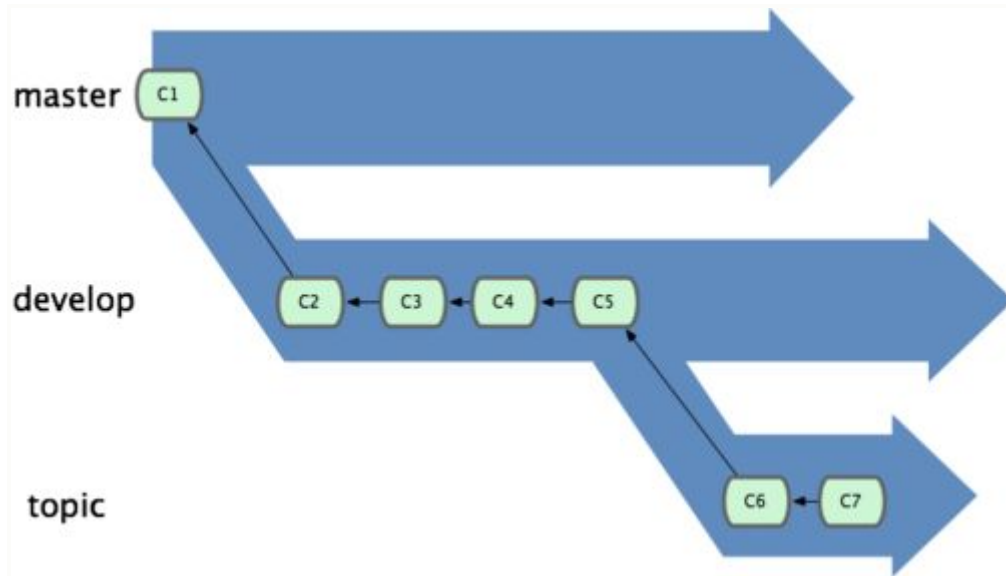


Figura 3-19. Puede ayudar pensar en las ramas como silos de almacenamiento.

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

### Ramas puntuales

Las ramas puntuales, en cambio, son útiles en proyectos de cualquier tamaño. Una rama puntual es aquella de corta duración que abres para un tema o para una funcionalidad muy concretos. Es algo que nunca habrías hecho en otro sistema VCS, debido a los altos costos de crear y fusionar ramas que se suelen dar en esos sistemas. Pero en Git, por el contrario, es muy habitual el crear, trabajar con, fusionar y borrar 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 unas pocas confirmaciones de cambio en ellas, y luego las has borrado tras fusionarlas con la rama principal. Esta técnica te posibilita realizar rápidos y completos saltos de contexto. 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. En lugar de verte obligado a fusionarlos en el orden en que fueron creados y 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 a una segunda rama 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 registro de confirmaciones (commit history) será algo parecido a la Figura 3-20.

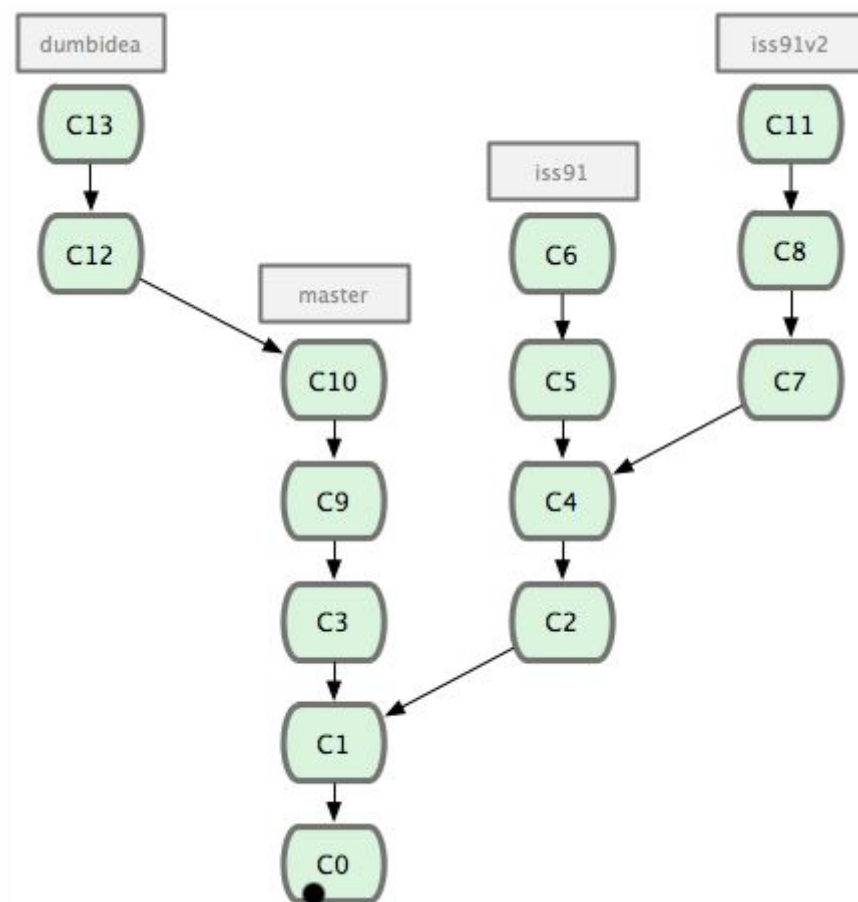


Figura 3-20. El registro de confirmaciones con múltiples ramas puntuales.

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 registro será algo parecido a la Figura 3-21.

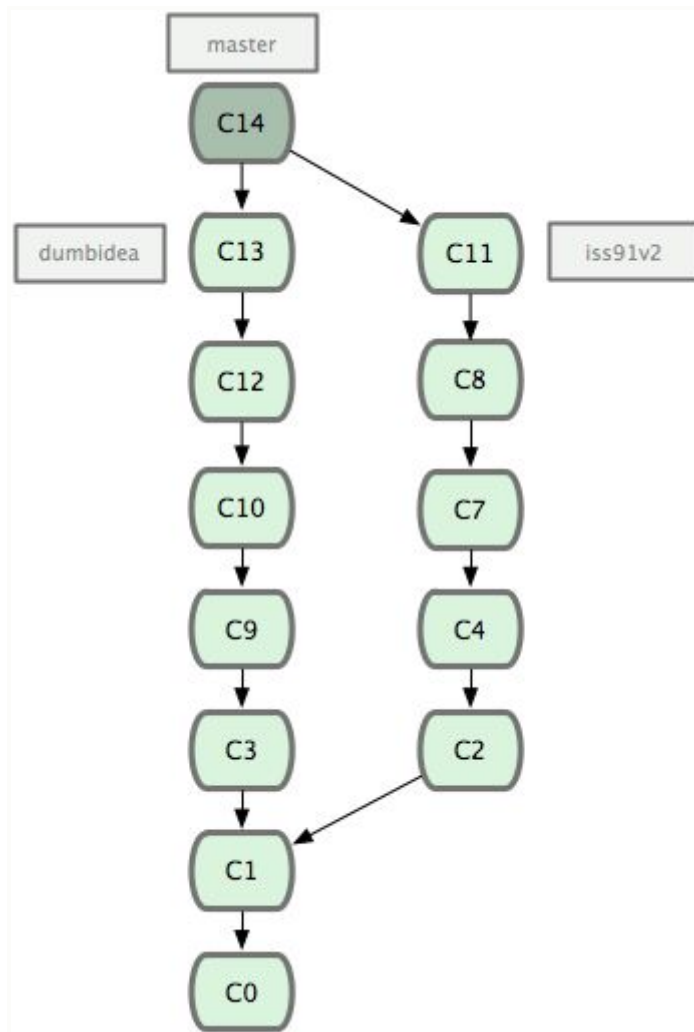


Figura 3-21. El registro tras fusionar **dumbidea** e **iss91v2**.

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 tráfico con ningún servidor.

## 3.5 Ramificaciones en Git - Ramas Remotas

## Ramas Remotas

Las ramas remotas son referencias al estado de 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**, tu tendrás tu propia rama de trabajo local **iss53**; pero la rama en el servidor apuntará a la última confirmación (commit) en la rama **origin/iss53**.

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

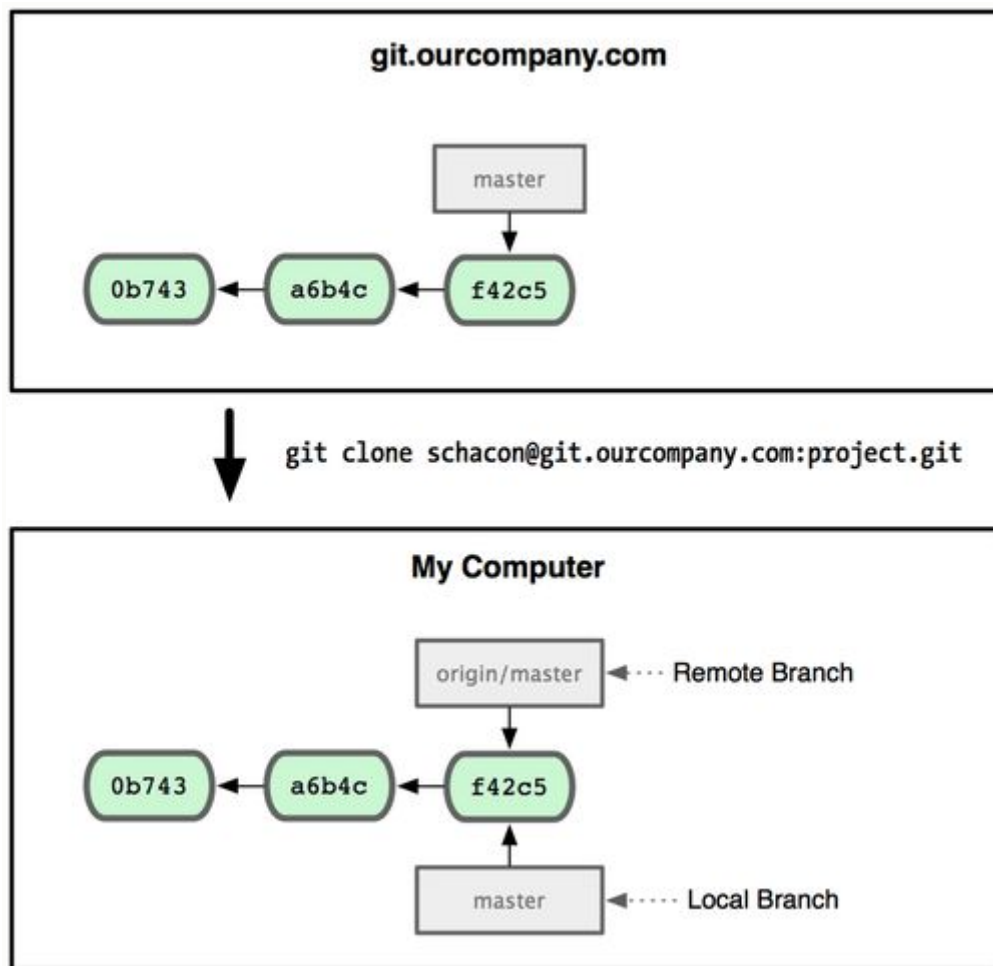


Figura 3-22. Un clón Git te proporciona tu propia rama `master` y otra rama `origin/master` apuntando a la rama `master` original.

Si haces algún trabajo en tu rama `master` local, y al mismo tiempo, alguna otra persona 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 (ver Figura 3/23).

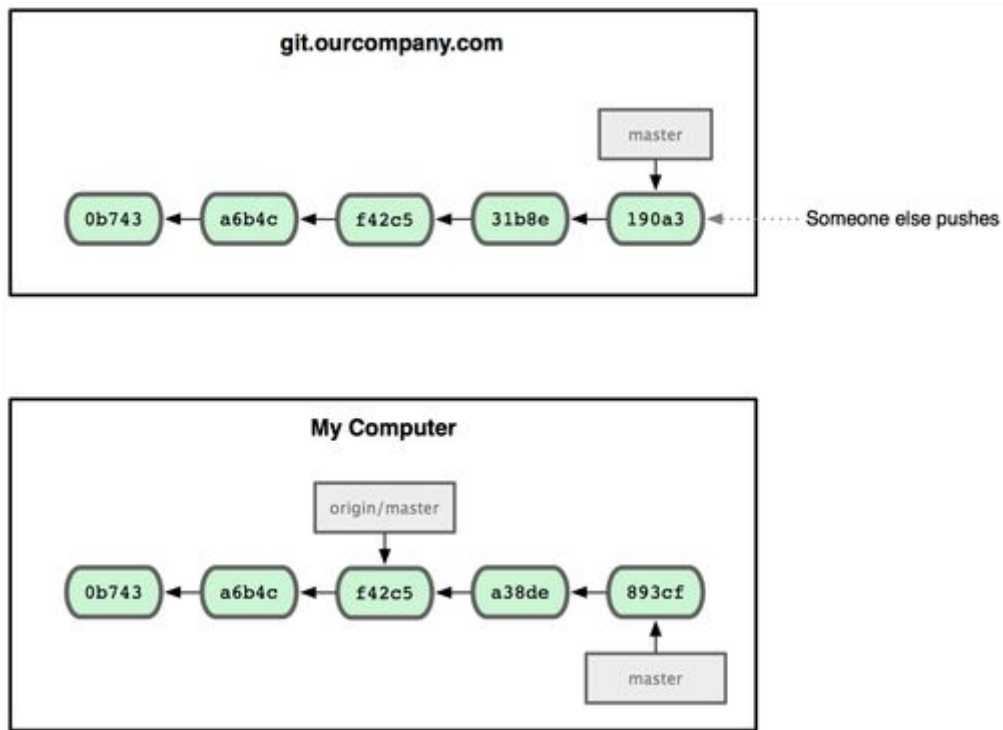


Figura 3-23. Trabajando localmente y que otra persona esté llevando (push) algo al servidor remoto, hace que cada registro avance de forma distinta.

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 tu no tengas, y actualiza tu base de datos local, moviendo tu rama `origin/master` para que apunte a esta nueva y más reciente posición (ver Figura 3-24).



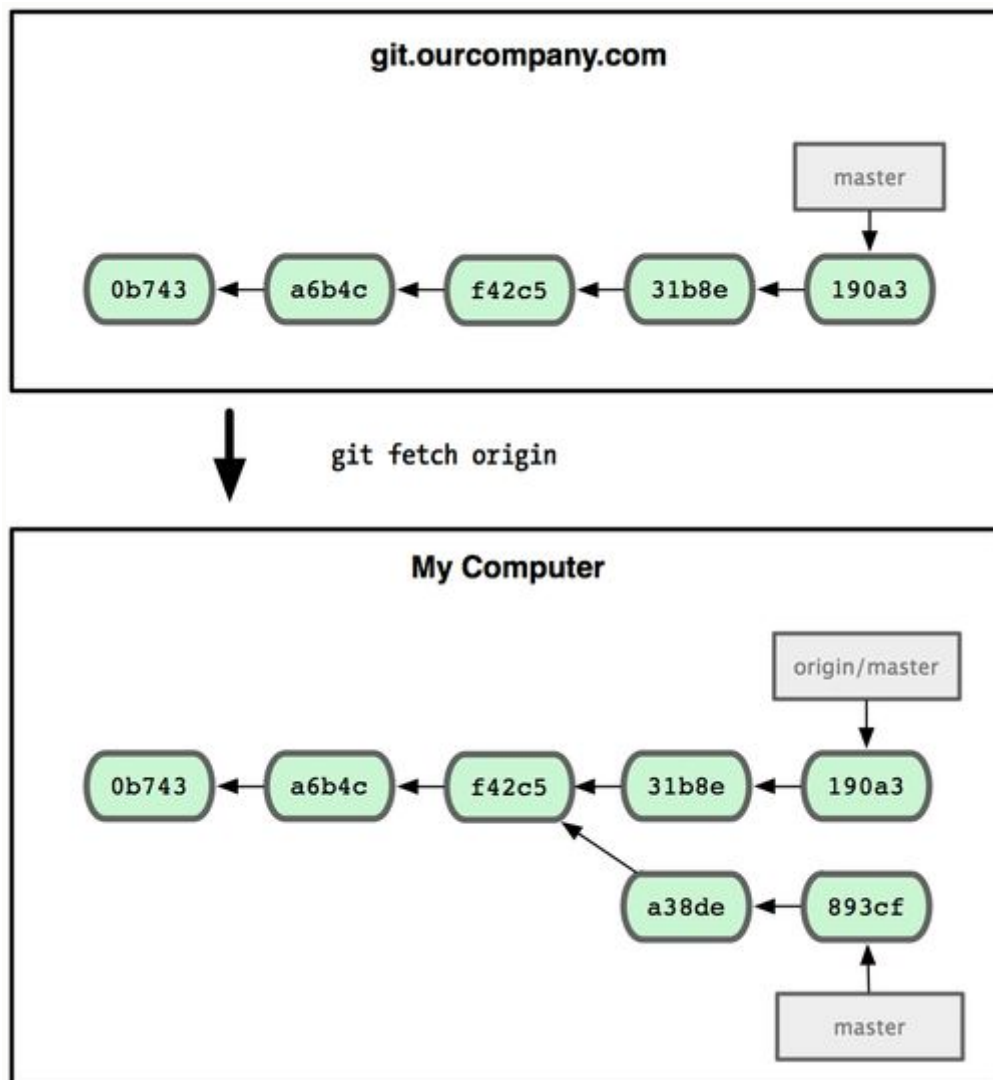


Figura 3-24. El comando `git fetch` actualiza tus referencias remotas.

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 solamente para desarrollo, por uno de tus equipos sprint. Un servidor 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 el capítulo 2. Puedes denominar `teamone` a este remoto, poniendo este nombre abreviado para la URL (ver Figura 3-25)

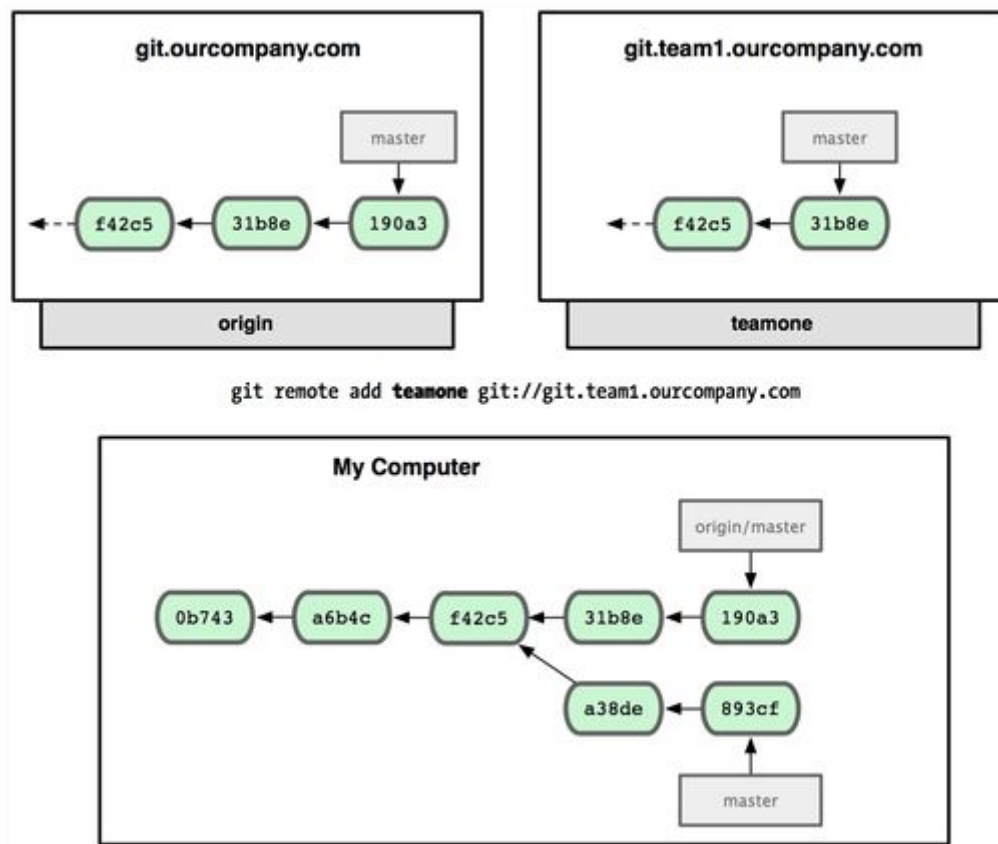


Figura 3-25. Añadiendo otro servidor como remoto.

Ahora, puedes usar el comando `git fetch teamone` para recuperar todo el contenido del servidor que tu no tenias. Debido a que dicho servidor es un subconjunto de 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`.

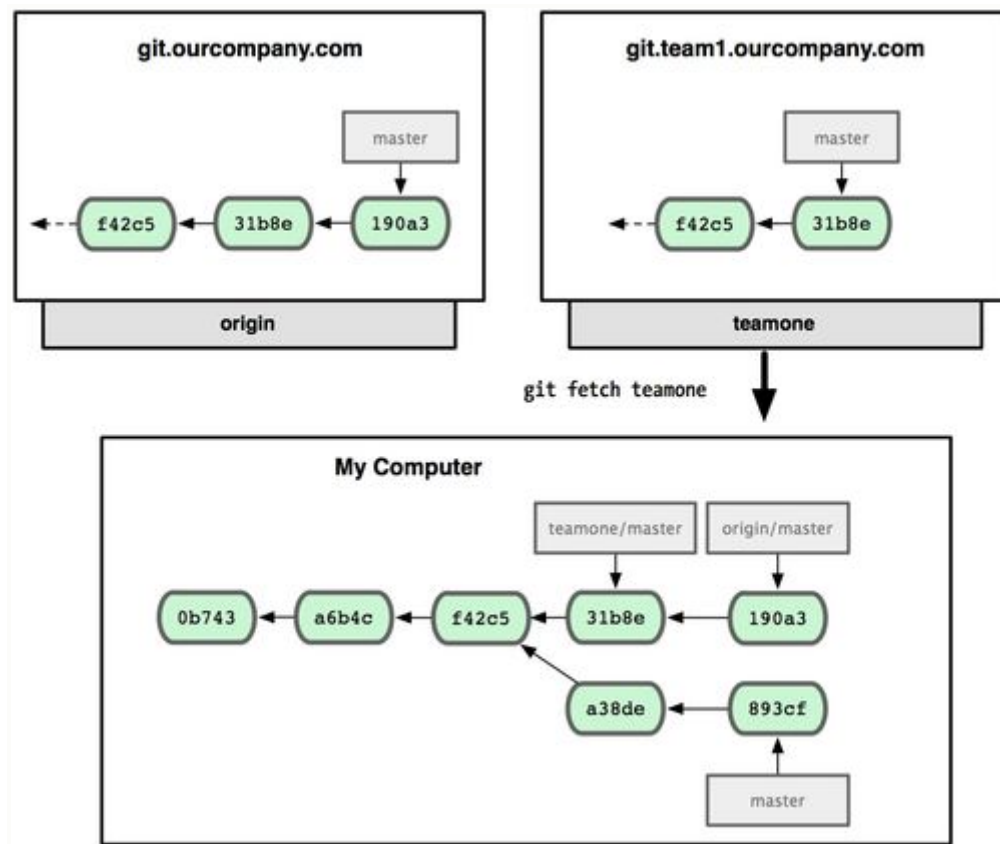


Figura 3-26. Obtienes una referencia local a la posición en la rama **master** de **teamone**.

## Publicando

Cuando quieres compartir una rama con el resto del mundo, has de 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 llevar (push) expresamente, cada vez, al remoto las ramas que desees compartir. De esta forma, puedes usar ramas privadas para el trabajo que no desees compartir. Llevando a un remoto tan solo aquellas partes que desees 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: 20, done.
Compressing objects: 100% (14/14), done.
Writing objects: 100% (15/15), 1.74 KiB, done.
Total 15 (delta 5), reused 0 (delta 0)
```

```
To git@github.com:schacon/simplegit.git
* [new branch]    serverfix -> serverfix
```

Esto es un poco como 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 el capítulo 9; aunque puedes ignorarlo por ahora. 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 otro nombre distinto. Si no quieres que se llame `serverfix` en el remoto, puedes lanzar, por ejemplo, `git push origin serverfix:awesomebranch`; para llevar tu rama `serverfix` local a la rama `awesomebranch` en el proyecto remoto.

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

```
$ git fetch origin
remote: Counting objects: 20, done.
remote: Compressing objects: 100% (14/14), done.
remote: Total 15 (delta 5), reused 0 (delta 0)
Unpacking objects: 100% (15/15), done.
From git@github.com:schacon/simplegit
* [new branch]    serverfix  -> origin/serverfix
```

Es importante destacar que cuando recuperas (fetch) nuevas ramas remotas, no obtienes automáticamente una copia editable local 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 actual rama de trabajo, puedes usar el comando `git merge origin/serverfix`. Y si quieres tener tu propia rama `serverfix`, donde puedas trabajar, puedes crearla directamente basándote en rama remota:

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

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

## Haciendo seguimiento a las ramas

Activando (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 push`, Git sabe automáticamente a qué servidor y a qué rama ha de llevar los contenidos. Igualmente, tecleando `git pull` mientras estamos en una de esas ramas, recupera (fetch) todas las referencias remotas y las consolida (merge) automáticamente en la correspondiente rama remota.

Cuando clonas un repositorio, este suele crear automáticamente una rama `master` que hace seguimiento de `origin/master`. Y es por eso que `git push` y `git pull` trabajan directamente, sin necesidad de más argumentos. Sin embargo, puedes preparar otras ramas de seguimiento si deseas tener unas que no hagan seguimiento de ramas en `origin` y que no sigan a la rama `master`. El ejemplo más simple, es el que acabas de ver al lanzar el comando `git checkout -b [rama] [nombreremoto]/[rama]`. Si tienes la versión 1.6.2 de Git, o superior, puedes utilizar también el parámetro `--track`:

```
$ git checkout --track origin/serverfix
Branch serverfix set up to track remote branch refs/remotes/origin/serverfix.
Switched to a new branch "serverfix"
```

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

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

Así, tu rama local `sf` va a llevar (push) y traer (pull) hacia o desde `origin/serverfix`.

## Borrando ramas remotas

Imagina que ya has terminado con una rama remota. Es decir, tanto tu como tus colaboradores habeis completado una determinada funcionalidad y la habeis incorporado (merge) a la rama `master` en el remoto (o donde quiera que tengais la rama de código estable). Puedes borrar la

rama remota utilizando la un tanto confusa sintaxis: `git push [nombreremoto] :[rama]`. Por ejemplo, si quieres borrar la rama `serverfix` del servidor, puedes utilizar:

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

Y... ¡Boom!. La rama en el servidor ha desaparecido. Puedes grabarte a fuego esta página, porque necesitarás ese comando y, lo más probable es que hayas olvidado su sintaxis. Una manera de recordar este comando es dándonos cuenta de que proviene de la sintaxis `git push [nombreremoto] [ramalocal]:[ramaremot]`. Si omites la parte `[ramalocal]`, lo que estás diciendo es: "no cojas nada de mi lado y haz con ello `[ramaremot]`".

## 3.6 Ramificaciones en Git - Reorganizando el trabajo realizado

### Reorganizando 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, como 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 (ver Figura 3-27), puedes ver que has separado tu trabajo y realizado confirmaciones (commit) en dos ramas diferentes.

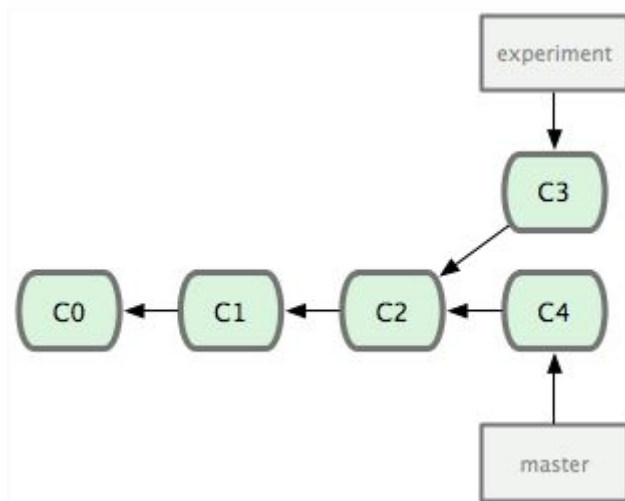


Figura 3-27. El registro de confirmaciones inicial.

La manera más sencilla de integrar ramas, tal y como hemos visto, es el comando **git merge**. Realiza una fusión a tres bandas entre las dos últimas instantáneas de cada rama (C3 y C4) y el ancestro común a ambas (C2); creando una nueva instantánea (snapshot) y la correspondiente confirmación (commit), según se muestra en la Figura 3-28.

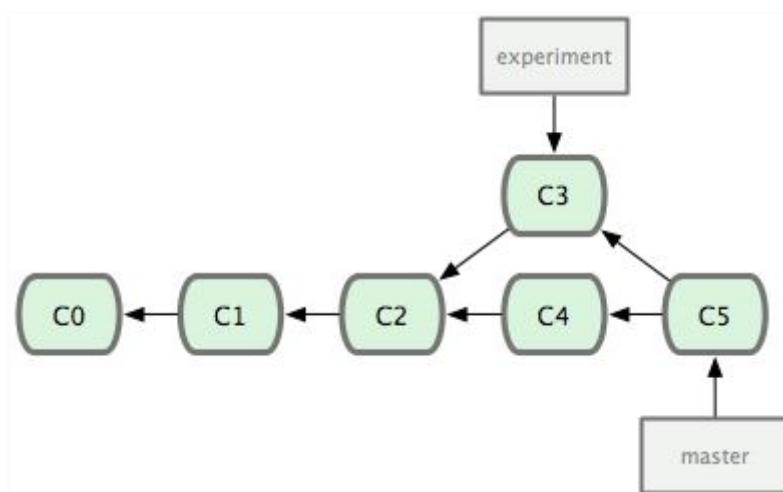


Figura 3-28. Fusionando una rama para integrar el registro de trabajos divergentes.

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

Por ejemplo, puedes lanzar los comandos:

\$ git checkout experiment

\$ git rebase master

First, rewinding head to replay your work on top of it...

Applying: added staged command

Haciendo que Git: vaya al ancestro común de ambas ramas (donde estás actualmente y de donde quieres reorganizar), saque las diferencias introducidas por cada confirmación en la rama donde estás, guarde esas diferencias en archivos temporales, reinicie (reset) la rama actual hasta llevarla a la misma confirmación en la rama de donde quieres reorganizar, y, finalmente, vuelva a aplicar ordenadamente los cambios. El proceso se muestra en la Figura 3-29.

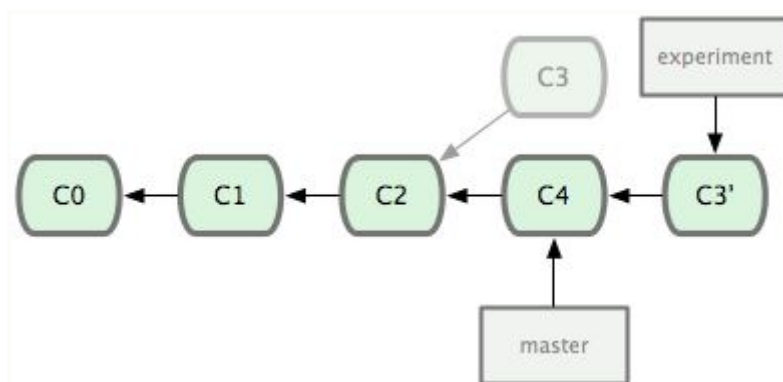


Figura 3-29. Reorganizando sobre C4 los cambios introducidos en C3.

En este momento, puedes volver a la rama **master** y hacer una fusión con avance rápido (fast-forward merge). (ver Figura 3-30)

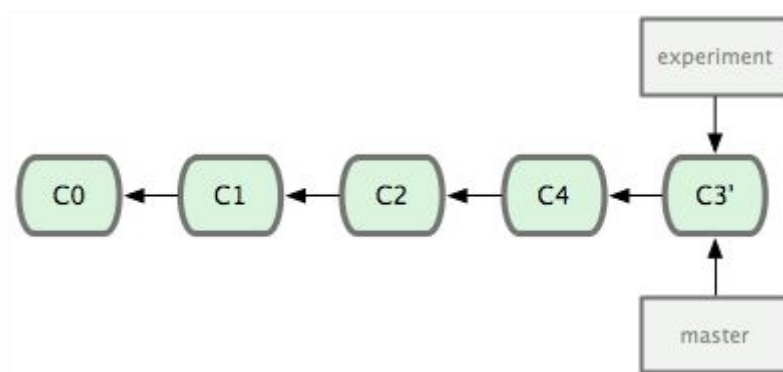


Figura 3-30. Avance rápido de la rama **master**.



Así, la instantánea apuntada por C3' aquí 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 registro más claro. Si examinas el registro de una rama reorganizada, este aparece siempre como un registro lineal: como si todo el trabajo se hubiera realizado en series, aunque realmente se haya hecho en paralelo.

Habitualmente, optarás por esta vía cuando quieras estar seguro de que tus confirmaciones de cambio (commits) se pueden aplicar limpiamente sobre una rama remota; posiblemente, en un proyecto donde estés intentando colaborar, pero lleves tu 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 registro. 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 otras reorganizaciones interesantes**

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

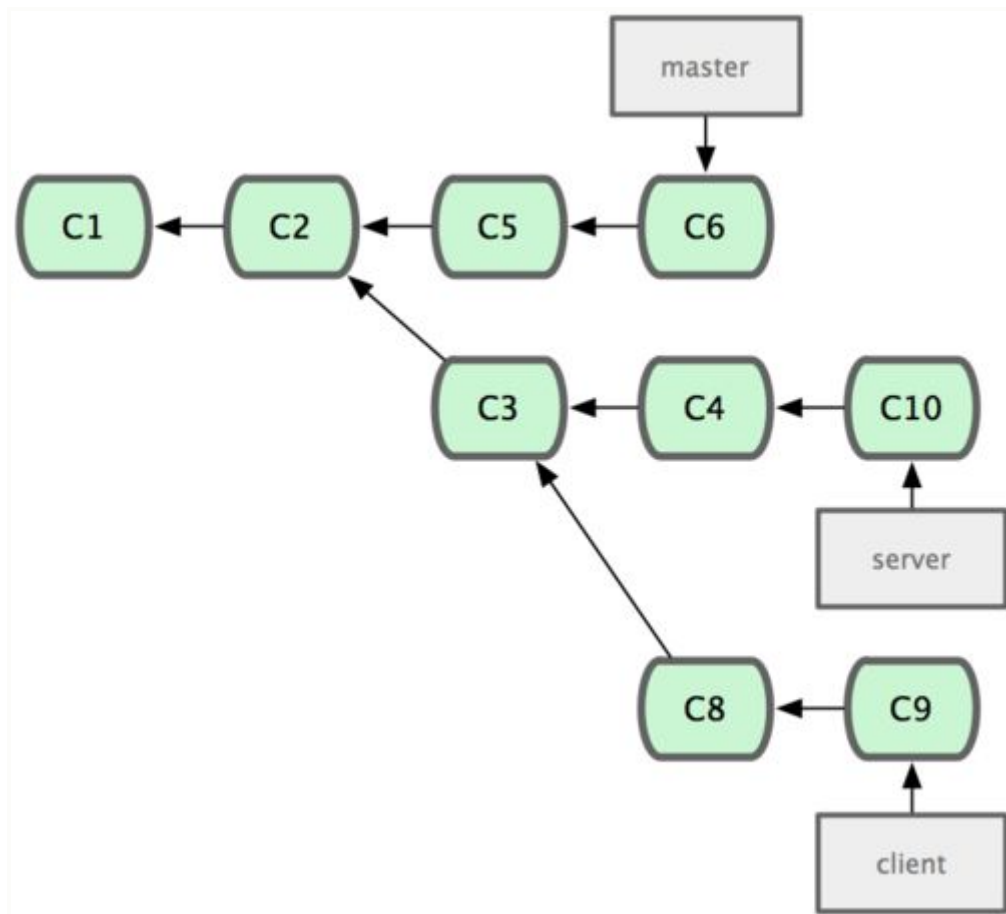


Figura 3-31. Un registro con una rama puntual sobre otra rama puntual.

Imagina que decides incorporar tus cambios de la parte cliente sobre el proyecto principal, para hacer un lanzamiento de versión; pero no quieres lanzar aún los cambios de la parte server 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 aplicarlos en la rama `master`. Puede parecer un poco complicado, pero los resultados, mostrados en la Figura 3-32, son realmente interesantes.



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 [ramabase] [ramapuntual]`. 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 la Figura 3-34.

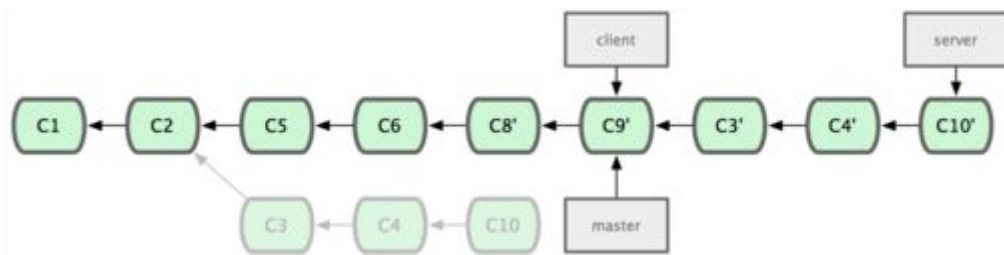


Figura 3-34. Reorganizando la rama **server** sobre la rama **branch**.

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 la Figura 3-35:

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

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

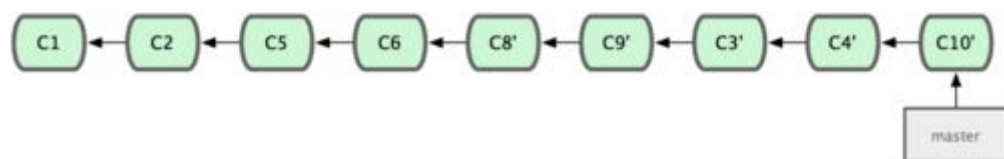


Figura 3-35. Registro final de confirmaciones de cambio.

## Los peligros de la reorganización

Ahh..., pero la dicha de la reorganización no la alcanzamos sin sus contrapartidas:

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

Siguiendo esta recomendación, no tendrás problemas. Pero si no la sigues, 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 tu y las reescribes con **git rebase** y las vuelves a enviar (push) de nuevo. Tus colaboradores tendrán que refusionar (re-merge) su trabajo y todo se volverá tremendamente complicado cuando intentes recoger (pull) su trabajo de vuelta sobre el tuyo.

Vamos a verlo con un ejemplo. Imagínate que haces un clon desde un servidor central, y luego trabajas sobre él. Tu registro de cambios puede ser algo como lo de la Figura 3-36.

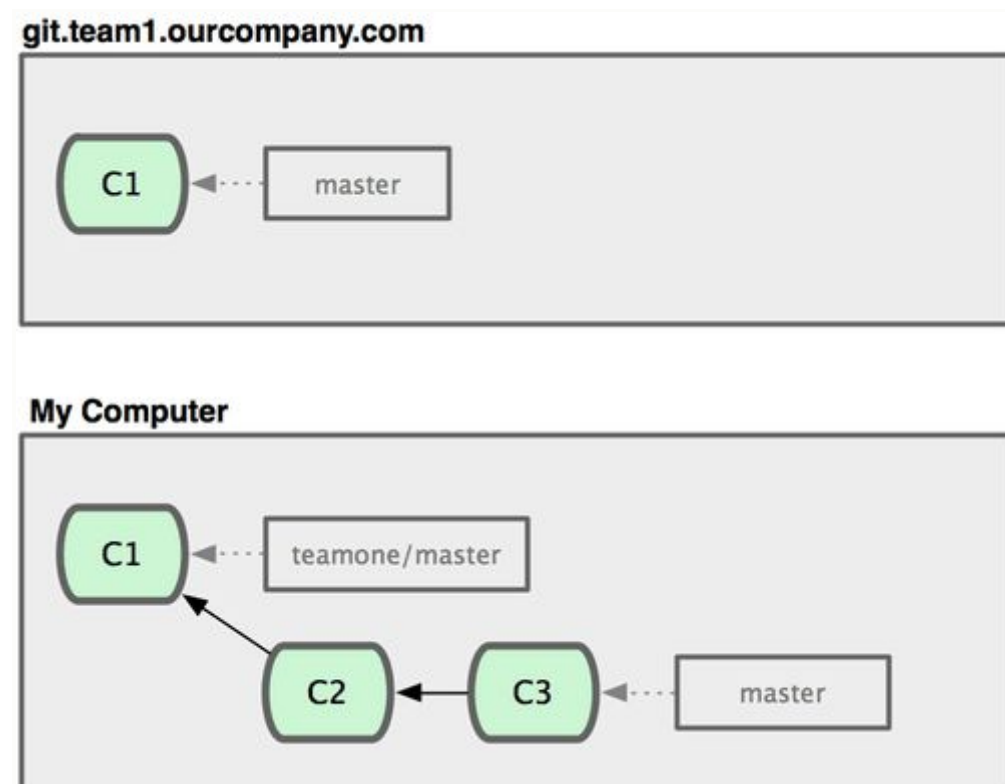


Figura 3-36. Clonar un repositorio y trabajar sobre él.

Ahora, otra persona trabaja también sobre ello, realiza una fusión (merge) y lleva (push) su trabajo al servidor central. Tu te traes (fetch) sus trabajos y los fusionas (merge) sobre una nueva rama en tu trabajo. Quedando tu registro de confirmaciones como en la Figura 3-37.

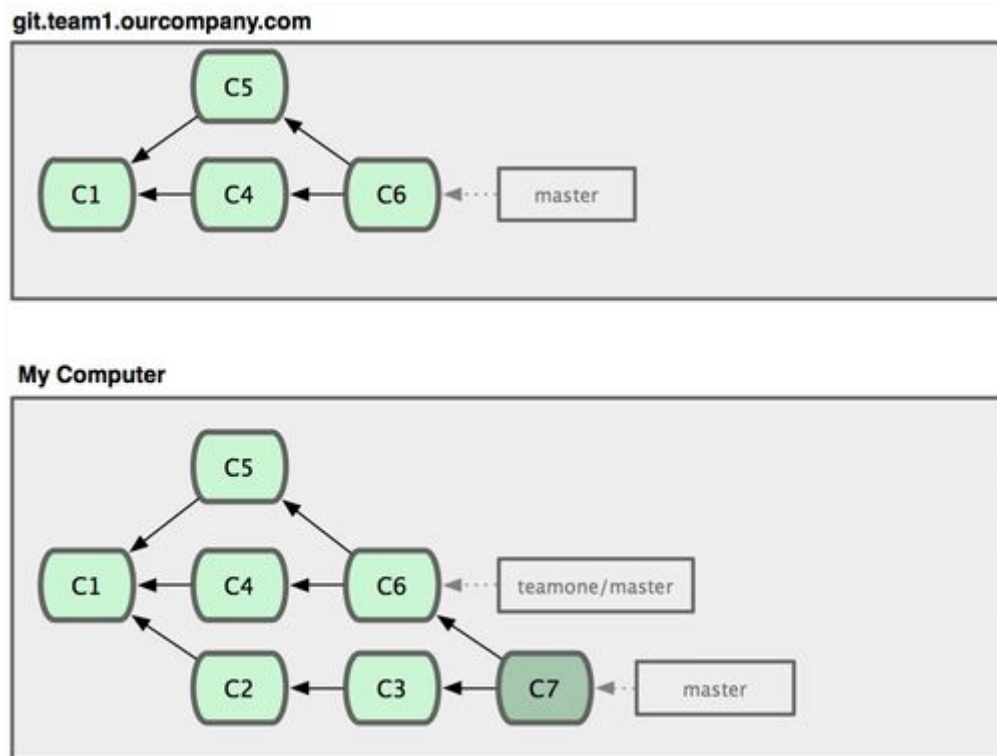


Figura 3-37. Traer (fetch) algunas confirmaciones de cambio (commits) y fusionarlas (merge) sobre tu trabajo.

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

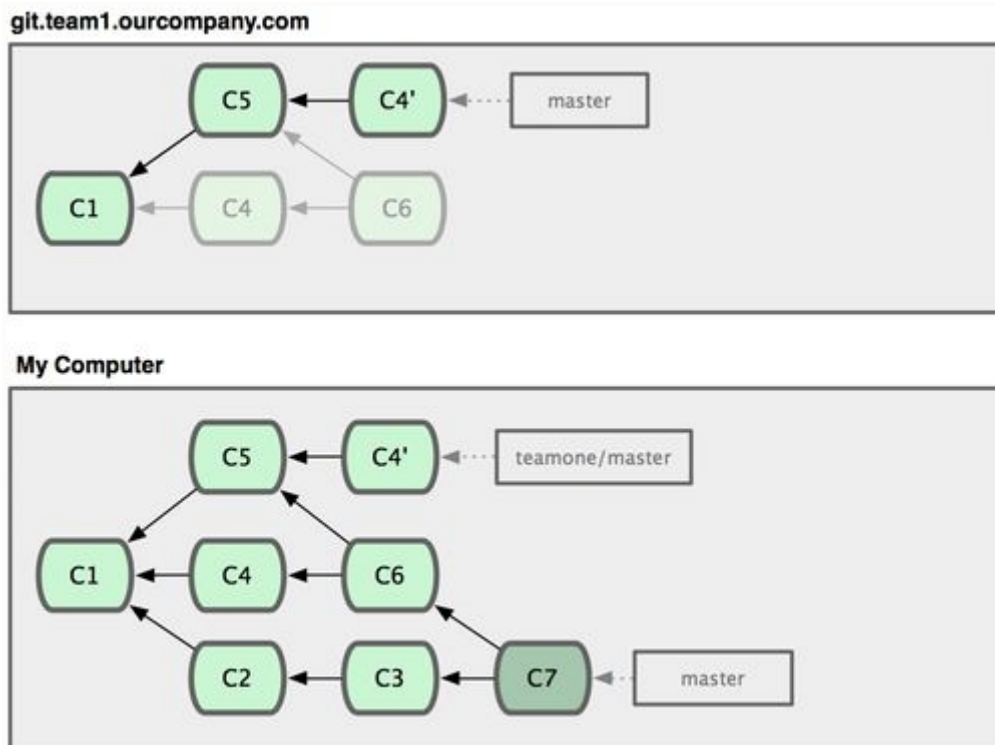


Figura 3-38. Alguien envié (push) confirmaciones (commits) reorganizadas, abandonando las confirmaciones en las que tu habías basado tu trabajo.

En ese momento, tu te ves obligado a fusionar (merge) tu trabajo de nuevo, aunque creías que ya lo habías hecho antes. La reorganización cambia los resúmenes (hash) SHA-1 de esas confirmaciones (commits), haciendo que Git se crea que son nuevas confirmaciones. Cuando realmente tu ya tenías el trabajo de C4 en tu registro.

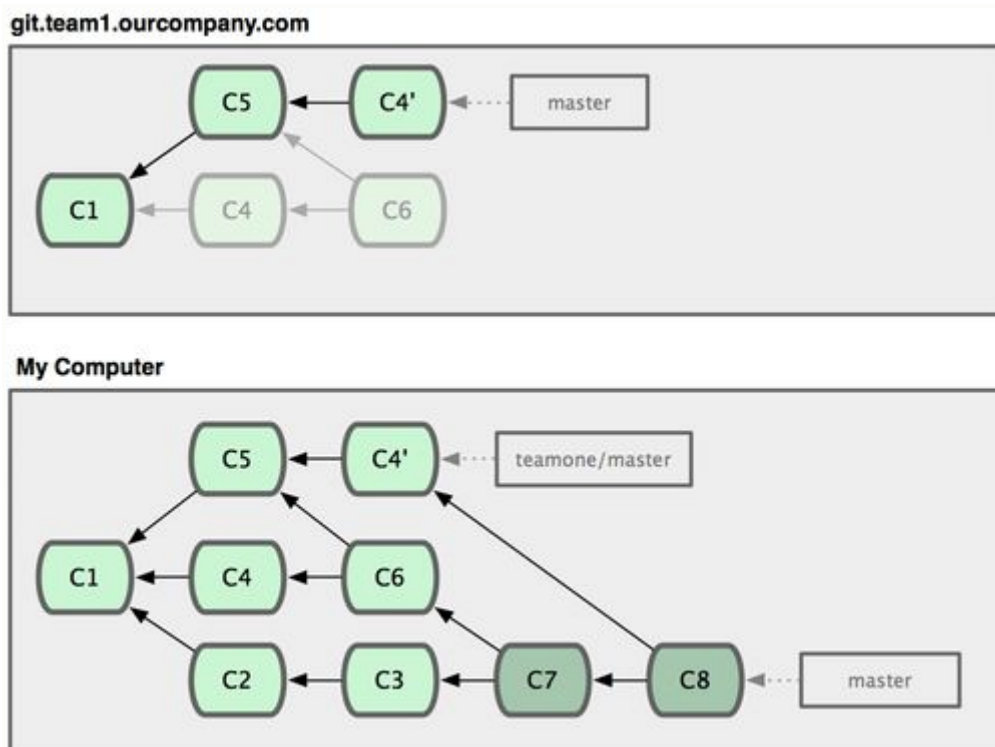


Figura 3-39. Vuelves a fusionar el mismo trabajo en una nueva fusión confirmada.

Te ves obligado a fusionar (merge) ese trabajo en algún punto, para poder seguir adelante con otros desarrollos en el futuro. Tras todo esto, tu registro de confirmaciones de cambio (commit history) contendrá tanto la confirmación C4 como la C4'; teniendo ambas el mismo contenido y el mismo mensaje de confirmación. Si lanzas un **git log** en un registro como este, verás dos confirmaciones con el mismo autor, misma fecha y mismo mensaje. Lo que puede llevar a confusiones. 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.

Si solo usas la reorganización como una vía para hacer limpieza y organizar confirmaciones de cambio antes de enviarlas, y si únicamente reorganizas confirmaciones que nunca han sido públicas. Entonces no tendrás problemas. Si, por el contrario, reorganizas confirmaciones que alguna vez han sido públicas y otra gente ha basado su trabajo en ellas. Entonces estarás en un aprieto.



# Chapter 4

## Git en un servidor

A estas alturas, ya podrás realizar la mayor parte de las tareas habituales trabajando con Git. Pero, para poder colaborar, necesitarás tener un repositorio remoto de Git. Aunque técnicamente es posible enviar (push) y recibir (pull) cambios directamente a o desde repositorios individuales, no es muy recomendable trabajar así por la gran facilidad de confundirte si no andas con sumo cuidado. Es más, si deseas que tus colaboradores puedan acceder a tu repositorio, incluso cuando tu ordenador este apagado, puede ser de gran utilidad disponer de un repositorio común fiable. En este sentido, el método más recomendable para colaborar con otra persona es preparar un repositorio intermedio donde ambos tengais acceso, enviando (push) y recibiendo (pull) a o desde allí. Nos referiremos a este repositorio como "servidor Git"; pero en seguida te darás cuenta de que solo se necesitan unos pocos recursos para albergar un repositorio Git, y, por tanto, no será necesario utilizar todo un servidor entero para él.

Disponer un servidor Git es simple. Lo primero, has de elegir el/los protocolo/s que deseas para comunicarte con el servidor. La primera parte de este capítulo cubrirá la gama de protocolos disponibles, detallando los pros y contras de cada uno de ellos. Las siguientes secciones explicarán algunas de las típicas configuraciones utilizando esos protocolos, y cómo podemos poner en marcha nuestro servidor con ellos. Por último, repasaremos algunas opciones albergadas on-line; por si no te preocupa guardar tu código en servidores de terceros y no deseas enredarte preparando y manteniendo tu propio servidor.

Si este es el caso, si no tienes interés de tener tu propio servidor, puedes saltar directamente a la última sección del capítulo; donde verás algunas opciones para dar de alta una cuenta albergada. Y después puedes moverte al capítulo siguiente, donde vamos a discutir algunos de los mecanismos para trabajar en un entorno distribuido.

Un repositorio remoto es normalmente un *repositorio básico mínimo*, un repositorio Git sin carpeta de trabajo. Debido a que dicho repositorio se va a utilizar exclusivamente como un punto de colaboración, no tiene sentido el tener una instantánea de trabajo (snapshot) activa en el disco (checkout); nos basta con tener solamente los propios datos Git. Básicamente, un repositorio básico mínimo son los contenidos de la carpeta '.git', tal cual, sin nada más.

## 4.1 Git en un servidor - Los Protocolos

### Los Protocolos

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

Merece destacar que, con la excepción del protocolo HTTP, todos los demás protocolos requieren que Git esté instalado y operativo en el servidor.

#### Protocolo Local

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 los casos en que todos se conectan al mismo ordenador. Aunque este último caso no es precisamente el ideal, ya que todas las instancias del repositorio estarían en la misma máquina; aumentando las posibilidades de una pérdida catastrófica.

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

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

O como:

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

Git trabaja ligeramente distinto si indicas 'file:/' de forma explícita al comienzo de la URL. Si escribes simplemente el camino, Git intentará usar enlaces rígidos (hardlinks) o copiar directamente los archivos que necesita. Si escribes con el prefijo 'file:/', Git lanza el proceso que usa habitualmente para transferir datos sobre una red; proceso que suele ser mucho menos

eficiente. La única razón que puedes tener para indicar expresamente el prefijo 'file:/' puede ser el querer una copia limpia del repositorio, descartando referencias u objetos superfluos.

Normalmente, tras haberlo importado desde otro sistema de control de versiones o algo similar (ver el Capítulo 9 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, "Disponiendo Git en un servidor", veremos cómo exportar un repositorio básico para conseguir esto.

Este camino es también útil para recuperar rápidamente el contenido del repositorio de trabajo de alguna otra persona. Si tu y otra persona estais trabajando en el mismo proyecto y ella 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.

## El Procotolo SSH

Probablemente, SSH sea el protocolo más habitual para Git. Debido a disponibilidad en la mayor parte de los servidores; (pero, si no lo estuviera disponible, además es sencillo habilitarlo). Por otro lado, SSH es el único protocolo de red con el que puedes facilmente tanto leer como escribir. Los otros dos protocolos de red (HTTP y Git) suelen ser normalmente protocolos de solo-lectura; de tal forma que, aunque los tengas disponibles para el público en general, sigues necesitando SSH para tu propio uso en escritura. Otra ventaja de SSH es el su mecanismo de autenticación, sencillo de habilitar y de usar.

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

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

O puedes prescindir del protocolo; Git asume SSH si no indicas nada expresamente: `$ 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, necesitas usarlo si quieres un acceso de escritura autenticado a tu repositorio. En segundo lugar, SSH es sencillo de habilitar. 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 procolos 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 procolo que tengas que emplear. Pero si quieres 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 sí que puedes habilitar el envío, si lo haces, dada la total falta de ningún mecanismo de autenticación, cualquiera que encuentre la URL a tu proyecto en Internet, podrá enviar (push) contenidos a él. Ni que decir tiene que esto solo lo necesitarás en contadas ocasiones.

## **Ventajas**

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

## **Desventajas**

La pega del protocolo Git, es su falta de autenticación. No es recomendable tenerlo como único protocolo de acceso a tus proyectos. Habitualmente, lo combinarás con un acceso SSH para los pocos desarrolladores con acceso de escritura que envíen (push) material. Usando 'git://' para los accesos solo-lectura del resto de personas. Por otro lado, es también el protocolo más complicado de implementar. Necesita activar su propio demonio, (tal y como se explica en la sección "Gitis", más adelante, en este capítulo); y necesita configurar 'xinetd' o similar, lo cual no suele estar siempre disponible en el sistema donde estés trabajando. Requiere además abrir expresamente acceso al puerto 9418 en el cortafuegos, ya que este no es uno de los puertos estándares que suelen estar habitualmente permitidos en los cortafuegos corporativos. Normalmente, este oscuro puerto suele estar bloqueado detrás de los cortafuegos corporativos.

## **El protocolo HTTP/S**

Por último, tenemos el protocolo HTTP. Cuya belleza radica en la simplicidad para habilitarlo. Basta con situar el repositorio Git bajo la raíz de los documentos HTTP y preparar el enganche

(hook) 'post-update' adecuado. (Ver el Capítulo 7 para detalles sobre los enganches Git.) A partir de ese momento, cualquiera con acceso al servidor web podrá clonar tu repositorio. Para permitir acceso a tu repositorio a través de HTTP, puedes hacer algo como esto:

```
$ 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 eso es todo. El enganche 'post-update' que viene de serie con Git se encarga de lanzar el comando adecuado ('git update-server-info') para hacer funcionar la recuperación (fetching) y el clonado (cloning) vía HTTP. Este comando se lanza automáticamente cuando envías (push) a este repositorio vía SSH; de tal forma que otras personas puedan clonarlo usando un comando tal que:

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

En este caso particular, estamos usando el camino '/var/www/htdocs', habitual en las configuraciones de Apache. Pero puedes utilizar cualquier servidor web estático, sin más que poner el repositorio en su camino. Los contenidos Git se sirven como archivos estáticos básicos (ver el Capítulo 9 para más detalles sobre servicios).

Es posible hacer que Git envíe (push) a través de HTTP. Pero no se suele usar demasiado, ya que requiere lidiar con los complejos requerimientos de WebDAV. Y precisamente porque se usa raramente, no lo vamos a cubrir en este libro. Si estás interesado en utilizar los protocolos HTTP-push, puedes encontrar más información en

<http://www.kernel.org/pub/software/scm/git/docs/howto/setup-git-server-over-http.txt>. La utilidad de habilitar Git para enviar (push) a través de HTTP es la posibilidad de utilizar cualquier servidor WebDAV para ello, sin necesidad de requerimientos específicos para Git. De tal forma que puedes hacerlo incluso a través de tu proveedor de albergue web, si este soporta WebDAV para escribir actualizaciones en tu sitio web.

## Ventajas

La mejor parte del protocolo HTTP es su sencillez de preparación. Simplemente lanzando unos cuantos comandos, dispones de un método sencillo de dar al mundo entero acceso a tu repositorio Git. En tan solo unos minutos. Además, el protocolo HTTP no requiere de grandes

recursos en tu servidor. Por utilizar normalmente un servidor HTTP estático, un servidor Apache estandar puede con un tráfico de miles de archivos por segundo; siendo difícil de sobrecargar incluso con el más pequeño de los servidores.

Puedes también servir tus repositorios de solo lectura a través de HTTPS, teniendo así las transferencias encriptadas. O puedes ir más lejos aún, requiriendo el uso de certificados SSL específicos para cada cliente. Aunque, si pretendes ir tan lejos, es más sencillo utilizar claves públicas SSH; pero ahí está la posibilidad, por si en algún caso concreto sea mejor solución el uso de certificados SSL u otros medios de autenticación HTTP para el acceso de solo-lectura a través de HTTPS.

Otro detalle muy útil de emplear HTTP, es que, al ser un protocolo de uso común, la mayoría de los cortafuegos corporativos suelen tener habilitado el tráfico a través de este puerto.

### Desventajas

La pega de servir un repositorio a través de HTTP es su relativa ineficiencia para el cliente. Suele requerir mucho más tiempo el clonar o el recuperar (fetch), debido a la mayor carga de procesamiento y al mayor volumen de transferencia que se da sobre HTTP respecto de otros protocolos de red. Y precisamente por esto, porque no es tan inteligente y no transfiere solamente los datos imprescindibles, (no hay un trabajo dinámico por parte del servidor), el protocolo HTTP suele ser conocido como el protocolo *estúpido*. Para más información sobre diferencias de eficiencia entre el protocolo HTTP y los otros protocolos, ver el Capítulo 9.

## 4.2 Git en un servidor - Poniendo Git en un Servidor

### Poniendo Git en un Servidor

El primer paso para preparar un servidor Git, es exportar un repositorio existente a un nuevo repositorio básico, a un repositorio sin carpeta de trabajo. Normalmente suele ser sencillo. Tan solo has de utilizar el comando 'clone' con la opción '--bare'. Por convenio, los nombres de los repositorios básicos suelen terminar en '.git', por lo que lanzaremos:

```
$ git clone --bare my_project my_project.git  
Initialized empty Git repository in /opt/projects/my_project.git/
```

El resultado de este comando es un poco confuso. Como 'clone' es fundamentalmente un 'git init' seguido de un 'git fetch', veremos algunos de los mensajes de la parte 'init', concretamente de la parte en que se crea una carpeta vacía. La copia de objetos no da ningún mensaje, pero también se realiza. Tras esto, tendrás una copia de los datos en tu carpeta 'my\_project.git'.

Siendo el proceso mas o menos equivalente a haber realizado:

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

Realmente hay un par de pequeñas diferencias en el archivo de configuración; pero, a efectos prácticos es casi lo mismo. Se coge el repositorio Git en sí mismo, sin la carpeta de trabajo, y se crea una copia en una nueva carpeta específica para él solo.

## Poniendo el repositorio básico en un servidor

Ahora que ya tienes una copia básica de tu repositorio, todo lo que te resta por hacer es colocarlo en un servidor y ajustar los protocolos. Supongamos que has preparado un servidor denominado 'git.example.com', con acceso SSH. Y que quieres guardar todos los repositorios Git bajo la carpeta '/opt/git'. Puedes colocar tu nuevo repositorio simplemente copiándolo:

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

A partir de entonces, cualquier otro usuario con acceso de lectura SSH a la carpeta '/opt/git' del servidor, podrá clonar el repositorio con la orden:

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

Y cualquier usuario SSH que tenga acceso de escritura a la carpeta '/opt/git/my\_project.git', tendrá también automáticamente acceso de volcado (push). Git añadirá automáticamente permisos de escritura al grupo sobre cualquier repositorio donde lances el comando 'git init' con la opción '--shared'.

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



Como se vé, es sencillo crear un repositorio básico a partir de un repositorio Git, y ponerlo en un servidor donde tanto tú como tus colaboradores tengais acceso SSH. Ahora ya estás preparado para trabajar con ellos en el proyecto común.

Es importante destacar que esto es, literalmente, todo lo necesario para preparar un servidor Git compartido. Habilitar unas cuantas cuentas SSH en un servidor; colocar un repositorio básico en algún lugar donde esos usuarios tengan acceso de lectura/escritura; y.... ¡ilisto!, eso es todo lo que necesitas.

En los siguientes apartados, se mostrará como ir más allá y preparar disposiciones más sofisticadas. Incluyendo temas tales como el evitar crear cuentas para cada usuario, el añadir acceso público de lectura, el disponer interfaces de usuario web, el usar la herramienta Gitis, y mucho más. Pero, ten presente que para colaborar con un pequeño grupo de personas en un proyecto privado, todo lo que necesitas es un servidor SSH y un repositorio básico.

## **Pequeños despliegues**

Si tienes un proyecto reducido o estás simplemente probando Git en tu empresa y sois unos pocos desarrolladores, el despliegue será sencillo. Porque la gestión de usuarios es precisamente uno de los aspectos más complicados de preparar un servidor Git. En caso de requerir varios repositorios de solo lectura para ciertos usuarios y de lectura/escritura para otros, preparar el acceso y los permisos puede dar bastante trabajo.

### **Acceso SSH**

Si ya dispones de un servidor donde todos los desarrolladores tengan acceso SSH, te será fácil colocar los repositorios en él (tal y como se verá en el próximo apartado). En caso de que necesites un control más complejo y fino sobre cada repositorio, puedes manejarlos a través de los permisos estandar del sistema de archivos.

Si deseas colocar los repositorios en un servidor donde no todas las personas de tu equipo tengan cuentas de acceso, tendrás que dar acceso SSH a aquellas que no lo tengan. Suponiendo que ya tengas el servidor, que el servicio SSH esté instalado y que sea esa la vía de acceso que tú estés utilizando para acceder a él.

Tienes varias maneras para dar acceso a todos los miembros de tu equipo. La primera forma es el habilitar cuentas para todos; es la manera más directa, pero también la más laboriosa. Ya que tendrías que lanzar el comando 'adduser' e inventarte contraseñas temporales para cada uno.

La segunda forma es el crear un solo usuario 'git' en la máquina y solicitar a cada persona que te envíe una clave pública SSH, para que puedas añadirlas al archivo `~/.ssh/authorized_keys` de

dicho usuario 'git'. De esta forma, todos pueden acceder a la máquina a través del usuario 'git'. Esto no afecta a los datos de las confirmaciones (commit), ya que el usuario SSH con el que te conectes no es relevante para las confirmaciones de cambios que registres.

Y una tercera forma es el preparar un servidor SSH autenticado desde un servidor LDAP o desde alguna otra fuente de autenticación externa ya disponible. Tan solo con que cada usuario pueda tener acceso al shell de la máquina, es válido cualquier mecanismo de autenticación SSH que se emplee para ello.

## 4.3 Git en un servidor - Generando tu clave pública SSH

### 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 no la tiene ya. El proceso para hacerlo es similar en casi cualquier sistema operativo. Ante todo, asegurarte que no tengas ya una clave. Por defecto, las claves de cualquier usuario SSH se guardan en la carpeta `~/.ssh` de dicho usuario. Puedes verificar si tienes ya unas claves, simplemente situandote sobre dicha carpeta y viendo su contenido:

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

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

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

```
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /Users/schacon/.ssh/id_rsa.
Your public key has been saved in /Users/schacon/.ssh/id_rsa.pub.
The key fingerprint is:
43:c5:5b:5f:b1:f1:50:43:ad:20:a6:92:6a:1f:9a:3a schacon@agadorlaptop.local
```

Como se vé, 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 este esté configurado con SSH y así lo requiera). Esto se puede realizar simplemente copiando los contenidos del archivo terminado en '.pub' y enviandoselos por correo electrónico. La clave pública será una serie de números, letras y signos, algo así como esto:

```
$ cat ~/.ssh/id_rsa.pub
ssh-rsa
AAAAB3NzaC1yc2EAAAABIwAAAQEAklOUpkDHrfHY17SbrmTIpNLTGK9Tjom/BWDSU
GPI+nafzIHDTYW7hdI4yZ5ew18JH4JW9jbhUFRviQzM7xIELEVf4h9IFX5QVkbPppSwg0c
da3
Pbv7kOdJ/MTyBIWXFCR+HAo3FXRitBqxiX1nKhXpHAZsMciLq8V6RjsNAQwdsdMFvSIV
K/7XA
t3FaoJoAsncM1Q9x5+3V0Ww68/eIFmb1zuUFljQJKprX88XypNDvjYNby6vw/Pb0rwert
/En
mZ+AW4OZPnTPI89ZPmVMLuayrD2cE86Z/il8b+gw3r3+1nKatmIkjn2so1d01QraTIMq
VSsbx
NrRFi9wrf+M7Q== schacon@agadorlaptop.local
```

Para más detalles sobre cómo crear unas claves SSH en variados sistemas operativos, consultar la correspondiente guía en GitHub: <http://github.com/guides/providing-your-ssh-key>.

## 4.4 Git en un servidor - Preparando el servidor

### Preparando 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 'claves autorizadas' para autenticar a tus usuarios. Se asume que tienes un servidor en marcha, con una distribución estandar de Linux, tal como Ubuntu. Comienzas creando un usuario 'git' y una carpeta '.ssh' para él.

```
$ sudo adduser git
$ su git
$ cd
$ mkdir .ssh
```

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
AAAAB3NzaC1yc2EAAAADAQABAAQCB007n/ww+ouN4gSLKssMxXnBOvf9LGt4L
ojG6rs6hPB09j9R/T17/x4lhJA0F3FR1rP6kYBRsWj2aThGw6HXLm9/5zytK6Ztg3RPPK+
4k
Yjh6541NYsnEAZuXz0jTTyAUfrtU3Z5E003C4oxOj6H0rfIF1kKI9MAQLMdpGW1GYEIgS
9Ez
Sdfd8AcCIicTDWbqLAcU4UpkaX8KyGILwsNuuGztobF8m72ALC/nLF6JLtPofwFBlgc+my
iv
O7TCUSBdLQlgMVOFq1I2uPWQOkOWQAHukEOMfjy2jctxSDBQ220ymjaNsHT4kgtZg2
AYYgPq
dAv8JggJICUvax2T9va5 gsg-keypair
```

No tienes más que añadirlas al archivo 'authorized\_keys':

```
$ 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 --bare init
```

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:

```
# en la máquina de John  
$ 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  
$ 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.

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':

```
$ sudo vim /etc/passwd
```

Localizar, al fondo, una línea parecida a:

```
git:x:1000:1000::/home/git:/bin/shgit:x:1000:1000::/home/git:/bin/sh
```

Y cambiar '/bin/sh' por '/usr/bin/git-shell' (nota: puedes utilizar el comando 'which git-shell' para ver dónde está instalado dicho shell). Quedará una línea algo así como:

```
git:x:1000:1000::/home/git:/usr/bin/git-shellgit:x:1000:1000::/home/git:/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: What do you think I am? A shell?
Connection to gitserver closed.
```

## 4.5 Git en un servidor - Acceso público

### Acceso público

¿Qué hacer si necesitas acceso anónimo de lectura a tu proyecto? Por ejemplo, si en lugar de albergar un proyecto privado interno, quieres albergar un proyecto de código abierto. O si tienes un grupo de servidores de integración automatizados o servidores de integración continua que cambian muy a menudo, y no quieres estar todo el rato generando claves SSH. Es posible que desees añadirles un simple acceso anónimo de lectura.

La manera más sencilla de hacerlo para pequeños despliegues, es el preparar un servidor web estático cuya raíz de documentos sea la ubicación donde tengas tus repositorios Git; y luego activar el anclaje (hook) 'post-update' que se ha mencionado en la primera parte de este capítulo.

Si utilizamos el mismo ejemplo usado anteriormente, suponiendo que tengas los repositorios en la carpeta '/opt/git', y que hay un servidor Apache en marcha en tu máquina. Veremos algunas configuraciones básicas de Apache, para que puedas hacerte una idea de lo que puedes necesitar. (Recordar que esto es solo un ejemplo, y que puedes utilizar cualquier otro servidor web.)

Lo primero, es activar el anclaje (hook):

```
$ cd project.git
$ mv hooks/post-update.sample hooks/post-update
$ chmod a+x hooks/post-update
```

Si utilizas una versión de Git anterior a la 1.6, el comando 'mv' no es necesario, ya que solo recientemente lleva Git los anclajes de ejemplo con el sufijo .sample

¿Que hace este anclaje 'post-update'? Pues tiene una pinta tal como:

```
$ cat .git/hooks/post-update
#!/bin/sh
exec git-update-server-info
```

Lo que significa que cada vez que envías (push) algo al servidor vía SSH, Git lanzará este comando y actualizará así los archivos necesarios para HTTP fetching. (ipendientedetraducir)

A continuación, has de añadir una entrada VirtualHost al archivo de configuración de Apache, fijando su raíz de documentos a la ubicación donde tengas tus proyectos Git. Aquí, estamos asumiendo que tienes un DNS comodín para redirigir \*.gitserver hacia cualquier máquina que estés utilizando para todo esto:

```
<VirtualHost *:80>
    ServerName git.gitserver
    DocumentRoot /opt/git
    <Directory /opt/git/>
        Order allow, deny
        allow from all
    </Directory>
</VirtualHost>
```

Asimismo, has de ajustar el grupo Unix de las carpetas bajo '/opt/git' a 'www-data', para que tu servidor web tenga acceso de lectura a los repositorios contenidos en ellas; porque la instancia de Apache que maneja los scripts CGI trabaja bajo dicho usuario:

```
$ chgrp -R www-data /opt/git
```

Una vez reinicies Apache, ya deberías ser capaz de clonar tus repositorios bajo dicha carpeta, simplemente indicando la URL de tu proyecto:

```
$ git clone http://git.gitserver/project.git
```

De esta manera, puedes preparar en cuestión de minutos accesos de lectura basados en HTTP a tus proyectos, para grandes cantidades de usuarios. Otra opción simple para habilitar accesos públicos sin autenticar, es arrancar el demonio Git, aunque esto supone demonizar el proceso. (Se verá esta opción en la siguiente sección.)

## 4.6 Git en un servidor - GitWeb

### 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. Puedes ver a GitWeb en acción en sitios como <http://git.kernel.org> (ver figura 4-1)





Figura 4-1. El interface web GitWeb.

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

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

Esto arranca un servidor HTTPD en el puerto 1234, y luego arranca un navegador que abre esa página. Es realmente sencillo. Cuando ya has 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 interface web para tu equipo o para un proyecto de código abierto que alberges, 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/gitweb.cgi
$ sudo cp -Rf gitweb /var/www/
```

Fijate 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. Por lo que si prefieres utilizar algún otro, no debería ser difícil de configurarlo. En este momento, deberías poder visitar 'http://gitserver/' para ver tus repositorios online. Y utilizar 'http://git.gitserver' para clonar (clone) y recuperar (fetch) tus repositorios a través de HTTP.

## 4.7 Git en un servidor - Gitis

### Gitis

Mantener claves públicas, para todos los usuarios, en el archivo 'authorized\_keys', puede ser una buena solución inicial. Pero, cuanto tengas cientos de usuarios, se hace bastante pesado gestionar así ese proceso. Tienes que iniciar sesión en el servidor cada vez. Y, además, no tienes control de acceso --todo el mundo presente en el archivo tiene permisos de lectura y escritura a todos y cada uno de los proyectos--.

En este punto, es posible que desees cambiar a un popular programa llamado Gitis. Gitis es básicamente un conjunto de scripts que te ayudarán a gestionar el archivo 'authorized\_keys', así como a implementar algunos controles de acceso simples. Lo interesante de la interfaz de usuario para esta herramienta de gestión de usuarios y de control de accesos, es que, en lugar de un interface web, es un repositorio especial de Git. Preparas la información en ese proyecto especial, y cuando la envías (push), Gitis reconfigura el servidor en base a ella. ¡Realmente interesante!.

Instalar Gitis no es precisamente sencillo. Pero tampoco demasiado complicado. Es más sencillo hacerlo si utilizas un servidor Linux --estos ejemplos se han hecho sobre un servidor Ubuntu 8.10--.

Gitis necesita de ciertas herramientas Python, por lo que la primera tarea será instalar el paquete de herramientas Pyton. En Ubuntu viene como el paquete python-setuptools:

```
$ sudo apt-get install python-setuptools
```

A continuación, has de clonar e instalar Gitis desde el repositorio principal de su proyecto:

```
$ git clone https://github.com/tv42/gitis.git  
$ cd gitis  
$ sudo python setup.py install
```

Esto instala un par de ejecutables, que serán los que Gitis utilice. Gitis intentará instalar sus repositorios bajo la carpeta '/home/git', lo cual está bien. Pero si, en lugar de en esa, has instalado tus repositorios bajo la carpeta '/opt/git'. Sin necesidad de reconfigurarlo todo, tan solo has de crear un enlace virtual:

```
$ ln -s /opt/git /home/git/repositories
```

Gitosis manejará tus claves por tí, por lo que tendrás que quitar el archivo actual, añadir de nuevo las claves más tarde, y dejar que Gitosis tome automáticamente el control del archivo 'authorizedkeys'. Para empezar, mueve el archivo 'authorizedkeys' a otro lado:

```
$ mv /home/git/.ssh/authorized_keys /home/git/.ssh/ak.bak
```

A continuación, restaura el inicio de sesión (shell) para el usuario 'git', (si es que lo habías cambiado al comando 'git-shell'). Los usuarios no podrán todavía iniciar sesión, pero Gitosis se encargará de ello. Así pues, cambia esta línea en tu archivo '/etc/passwd':

```
git:x:1000:1000::/home/git:/usr/bin/git-shellgit:x:1000:1000::/home/git:/usr/bin/git-shell
```

de vuelta a:

```
git:x:1000:1000::/home/git:/bin/shgit:x:1000:1000::/home/git:/bin/sh
```

Y, en este punto, ya podemos inicializar Gitosis. Lo puedes hacer lanzando el comando 'gitosis-init' con tu clave pública personal. Si tu clave pública personal no está en el servidor, la has de copiar a él:

```
$ sudo -H -u git gitosis-init < /tmp/id_dsa.pub
Initialized empty Git repository in /opt/git/gitosis-admin.git/
Reinitialized existing Git repository in /opt/git/gitosis-admin.git/
```

Esto habilita al usuario con dicha clave pública para que pueda modificar el repositorio principal de Git, y, con ello, pueda controlar la instalación de Gitosis. A continuación, has de ajustar manualmente el bit de ejecución en el script 'post-update' de tu nuevo repositorio de control:

```
$ sudo chmod 755 /opt/git/gitosis-admin.git/hooks/post-update
```

Y ya estás preparado para trabajar. Si lo has configurado todo correctamente, puedes intentar conectarte, vía SSH, a tu servidor como el usuario con cuya clave pública has inicializado Gitosis. Y deberás ver algo así como esto:

```
$ ssh git@gitserver
PTY allocation request failed on channel 0
fatal: unrecognized command 'gitosis-serve schacon@quaternion'
Connection to gitserver closed.
```

Indicandote que Gitosis te ha reconocido, pero te está hechando debido a que no estás intentando lanzar ningún comando Git. Por tanto, intentalo con un comando Git real --por ejemplo, clonar el propio repositorio de control de Gitosis a tu ordenador personal--

```
$ git clone git@gitserver:gitosis-admin.git
```

Con ello, tendrás una carpeta denominada 'gitosis-admin', con dos partes principales dentro de ella:

```
$ cd gitosis-admin
$ find .
./gitosis.conf
./keydir
./keydir/scott.pub
```

El archivo 'gitosis.conf' es el archivo de control que usarás para especificar usuarios, repositorios y permisos. La carpeta 'keydir' es donde almacenarás las claves públicas para los usuarios con acceso a tus repositorios --un archivo por usuario--. El nombre del archivo en la carpeta 'keydir' ('scott.pub' en el ejemplo), puede ser diferente en tu instalación, (Gitosis lo obtiene a partir de la descripción existente al final de la clave pública que haya sido importada con el script 'gitosis-init').

Si miras dentro del archivo 'gitosis.conf', encontrarás únicamente información sobre el proyecto 'gitosis-admin' que acabas de clonar:

```
$ cat gitosis.conf
[gitosis]

[group gitosis-admin]
writable = gitosis-admin
members = scott
```

Indicando que el usuario 'scott' --el usuario con cuya clave pública se ha inicializado Gitis-- es el único con acceso al proyecto 'gitis-admin'.

A partir de ahora, puedes añadir nuevos proyectos. Por ejemplo, puedes añadir una nueva sección denominada 'mobile', donde poner la lista de los desarrolladores en tu equipo móvil y los proyectos donde estos vayan a trabajar. Por ser 'scott' el único usuario que tienes definido por ahora, lo añadirás como el único miembro. Y puedes crear además un proyecto llamado 'iphone\_project' para empezar:

```
[group mobile]
writable = iphone_project
members = scott
```

Cada cambio en el proyecto 'gitis-admin', lo has de confirmar (commit) y enviar (push) de vuelta al servidor, para que tenga efecto sobre él:

```
$ git commit -am 'add iphone_project and mobile group'
[master]: created 8962da8: "changed name"
 1 files changed, 4 insertions(+), 0 deletions(-)
$ git push
Counting objects: 5, done.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 272 bytes, done.
Total 3 (delta 1), reused 0 (delta 0)
To git@gitserver:/opt/git/gitis-admin.git
 fb27aec..8962da8  master -> master
```

Puedes crear tu nuevo proyecto 'iphone\_project' simplemente añadiendo tu servidor como un remoto a tu versión local del proyecto de control y enviando (push). Ya no necesitarás crear manualmente repositorios básicos vacíos para los nuevos proyectos en el servidor. Gitis se encargará de hacerlo por tí, en cuanto realices el primer envío (push) de un nuevo proyecto:

```
$ git remote add origin git@gitserver:iphone_project.git
$ git push origin master
Initialized empty Git repository in /opt/git/iphone_project.git/
Counting objects: 3, done.
Writing objects: 100% (3/3), 230 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
To git@gitserver:iphone_project.git
```

```
* [new branch]    master -> master
```

Ten en cuenta que no es necesario indicar expresamente un camino (path), --de hecho, si lo haces, no funcionará--. Simplemente, has de poner un punto y el nombre del proyecto, --Gitosis se encargará de encontrarlo--.

Si deseas compartir el proyecto con tus compañeros, tienes que añadir de nuevo sus claves públicas. Pero en lugar de hacerlo manualmente sobre el archivo `~/.ssh/authorized_keys` de tu servidor, has de hacerlo --un archivo por clave-- en la carpeta 'keydir' del proyecto de control. Según pongas los nombres a estos archivos, así tendrás que referirte a los usuarios en el archivo 'gitosis.conf'. Por ejemplo, para añadir las claves públicas de John, Josie y Jessica:

```
$ cp /tmp/id_rsa.john.pub keydir/john.pub
$ cp /tmp/id_rsa.josie.pub keydir/josie.pub
$ cp /tmp/id_rsa.jessica.pub keydir/jessica.pub
```

Y para añadirlos al equipo 'mobile', dándoles permisos de lectura y escritura sobre el proyecto 'phone\_project':

```
[group mobile]
writable = iphone_project
members = scott john josie jessica
```

Tras confirmar (commit) y enviar (push) estos cambios, los cuatro usuarios podrán acceder a leer y escribir sobre el proyecto.

Gitosis permite también sencillos controles de acceso. Por ejemplo, si quieres que John tenga únicamente acceso de lectura sobre el proyecto, puedes hacer:

```
[group mobile]
writable = iphone_project
members = scott josie jessica
```

```
[group mobile_ro]
readonly = iphone_project
members = john
```

Habilitándole así para clonar y recibir actualizaciones desde el servidor; pero impidiéndole enviar de vuelta cambios al proyecto. Puedes crear tantos grupos como desees, para diferentes usuarios

y proyectos. También puedes indicar un grupo como miembro de otro (utilizado el prefijo '@'), para incluir todos sus miembros automáticamente:

```
[group mobile_committers]
members = scott josie jessica
```

```
[group mobile]
writable = iphone_project
members = @mobile_committers
```

```
[group mobile_2]
writable = another_iphone_project
members = @mobile_committers john
```

Si tienes problemas, puede ser útil añadir `loglevel=DEBUG` en la sección `[gitosis]`. Si, por lo que sea, pierdes acceso de envío (push) de nuevos cambios, (por ejemplo, tras haber enviado una configuración problemática); siempre puedes arreglar manualmente, en el propio servidor, el archivo `'/home/git/.gitosis.conf'`, (el archivo del que Gitosis lee su configuración). Un envío (push) de cambios al proyecto, coge el archivo `'gitosis.conf'` enviado y sobrescribe con él el del servidor. Si lo editas manualmente, permanecerá como lo dejes; hasta el próximo envío (push) al proyecto `'gitosis-admin'`.

## 4.8 Git en un servidor - El demonio Git

### El demonio Git

Para dar a tus proyectos un acceso público, sin autenticar, de solo lectura, querrás ir más allá del protocolo HTTP y comenzar a utilizar el protocolo Git. Principalmente, por razones de velocidad. El protocolo Git es mucho más eficiente y, por tanto, más rápido que el protocolo HTTP. Utilizándolo, ahorrarás mucho tiempo a tus usuarios.

Aunque, sigue siendo solo para acceso únicamente de lectura y sin autenticar. Si lo estás utilizando en un servidor fuera del perímetro de tu cortafuegos, se debe utilizar exclusivamente para proyectos que han de ser públicos, visibles para todo el mundo. Si lo estás utilizando en un



servidor dentro del perímetro de tu cortafuegos, puedes utilizarlo para proyectos donde un gran número de personas o de ordenadores (integración continua o servidores de desarrollo) necesiten acceso de solo lectura. Y donde quieras evitar la gestión de claves SSH para cada una de ellas.

En cualquier caso, el protocolo Git es relativamente sencillo de configurar. Tan solo necesitas lanzar este comando de forma demonizada:

```
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' con el que hemos lanzado Gitis.

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 en tu servidor Gitis a cuales de tus repositorios ha de permitir acceso sin autentificar por parte del servidor Git. Añadiendo una sección por cada repositorio, puedes indicar a cuáles permitirá leer el demonio Git. Por ejemplo, si quieres permitir acceso a tu 'proyecto iphone', puedes añadir lo siguiente al archivo 'gitis.conf':

```
[repo iphone_project]
daemon = yes
```

Cuando confirmes (commit) y envíes (push) estos cambios, el demonio que está en marcha en el servidor comenzará a responder a peticiones de cualquiera que solicite dicho proyecto a través del puerto 9418 de tu servidor.

Si decides no utilizar Gitis, pero sigues queriendo utilizar un demonio Git, has de lanzar este comando en cada proyecto que desees servir vía el demonio Git:

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

La presencia de este archivo, indica a Git que está permitido el servir este proyecto sin necesidad de autentificación.

También podemos controlar a través de Gitis los proyectos a ser mostrados por GitWeb.

Previamente, has de añadir algo como esto al archivo '/etc/gitweb.conf':

```
$projects_list = "/home/git/gitis/projects.list";
$projectroot = "/home/git/repositories";
$export_ok = "git-daemon-export-ok";
@git_base_url_list = ('git://gitserver');
```

Los proyectos a ser mostrados por GitWeb se controlarán añadiendo o quitando parámetros 'gitweb' en el archivo de configuración de Gitis. Por ejemplo, si quieres mostrar el proyecto iphone, has de poner algo así como:

```
[repo iphone_project]
daemon = yes
gitweb = yes
```

A partir de ese momento, cuando confirmes cambios (commit) y envíes (push) el proyecto, GitWeb comenzará a mostrar tu proyecto iphone.

## 4.9 Git en un servidor - Git en un alojamiento externo

### Git en un alojamiento externo

Si no quieres realizar todo el trabajo de preparar tu propio servidor Git, tienes varias opciones para alojar tus proyectos Git en una ubicación externa dedicada. Esta forma de trabajar tiene varias ventajas: un albergue externo suele ser rápido de configurar y sencillo de iniciar proyectos en él; además de no ser necesario preocuparte de su mantenimiento ni de su monitorización. Incluso en el caso de que tengas tu propio servidor interno, puede resultar interesante utilizar también un lugar público; para albergar tu código abierto --normalmente, ahí suele ser más sencillo de localizar por parte de la comunidad--

Actualmente tienes un gran número de opciones del alojamiento, cada una con sus ventajas y desventajas. Para obtener una lista actualizada, puedes mirar en la página GitHosting del wiki principal de Git:

<https://git.wiki.kernel.org/index.php/GitHosting>

Por ser imposible el cubrir todos ellos, y porque da la casualidad de que trabajo en uno de ellos, concretamente, en esta sección veremos cómo crear una cuenta y nuevos proyectos albergados en 'GitHub'. Así podrás hacerte una idea de cómo suelen funcionar estos alberges externos.

GitHub es, de lejos, el mayor sitio de albergue público de proyectos Git de código abierto. Y es también uno de los pocos que ofrece asimismo opciones de albergue privado; de tal forma que puedes tener tanto tus proyectos de código abierto y como los de código comercial cerrado en un

mismo emplazamiento. De hecho, nosotros utilizamos también GitHub para colaborar privadamente en este libro.

## GitHub

GitHub es ligeramente distinto a otros sitios de alberge, en tanto en cuanto que contempla espacios de nombres para los proyectos. En lugar de estar focalizado en los proyectos, GitHub gira en torno a los usuarios. Esto significa que, cuando alojo mi proyecto 'grit' en GitHub, no lo encontraras bajo 'github.com/grit', sino bajo 'github.com/schacon/grit'. No existe una versión canónica de ningún proyecto, lo que permite a cualquiera de ellos ser movido facilmente de un usuario a otro en el caso de que el primer autor lo abandone.

GitHub es también una compañía comercial, que cobra por las cuentas que tienen repositorios privados. Pero, para albergar proyectos públicos de código abierto, cualquiera puede crear una cuenta gratuita. Vamos a ver cómo hacerlo.

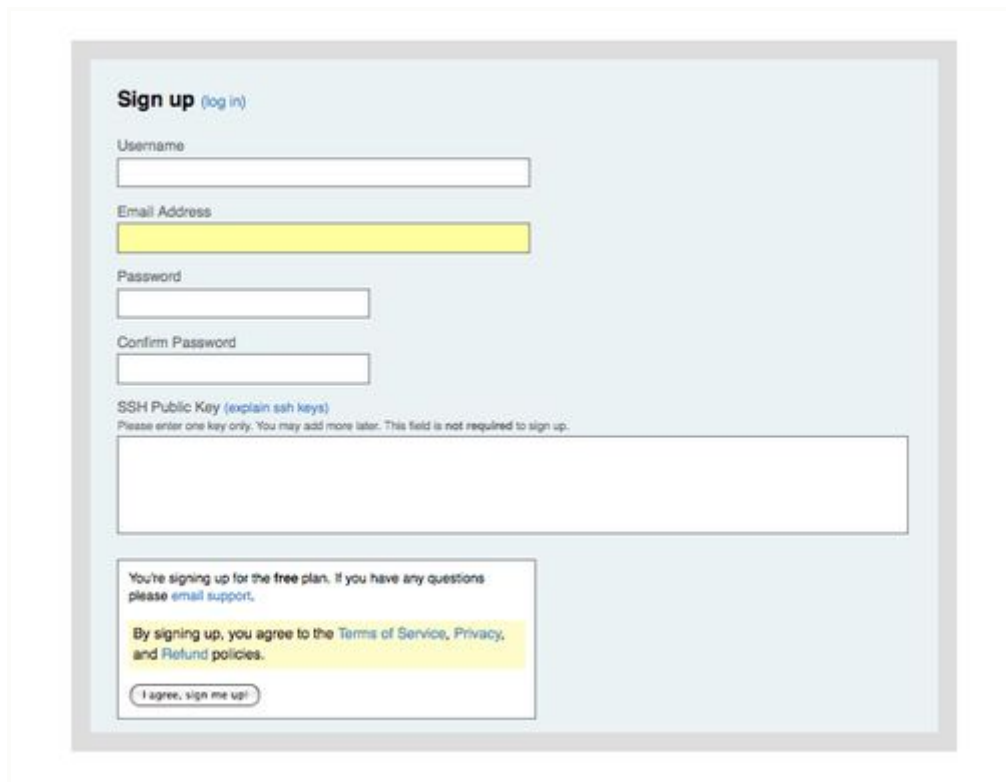
## Configurando una cuenta de usuario

El primer paso es dar de alta una cuenta gratuita. Si visitas la página de Precios e Inicio de Sesión, en 'https://github.com/pricing', y clicas sobre el botón "Registro" ("Sign Up") de las cuentas gratuitas, verás una página de registro:



Figura 4-2. La página de planes GitHub.

En ella, has de elegir un nombre de usuario que esté libre, indicar una cuenta de correo electrónico y poner una contraseña.



The image shows the GitHub sign-up form. At the top, it says "Sign up (log in)". Below this are input fields for "Username", "Email Address", "Password", and "Confirm Password". There is a large text area for "SSH Public Key" with a link to "explain ssh keys" and a note: "Please enter one key only. You may add more later. This field is not required to sign up." At the bottom, there is a box containing the text: "You're signing up for the free plan. If you have any questions please email support." Below this is a yellow box with the text: "By signing up, you agree to the Terms of Service, Privacy, and Refund policies." At the very bottom is a button that says "I agree, sign me up!"

Figura 4-3. El formulario de registro en GitHub.

Si la tuvieras, es también un buen momento para añadir tu clave pública SSH. Veremos cómo generar una de estas claves, más adelante, en la sección "Ajustes Simples". Pero, si ya tienes un par de claves SSH, puedes coger el contenido correspondiente a la clave pública y pegarlo en la caja de texto preparada para tal fin. El enlace "explicar claves ssh" ("explain ssh keys") te llevará a unas detalladas instrucciones de cómo generarlas en la mayor parte de los principales sistemas operativos. Clicando sobre el botón de "Estoy de acuerdo, registramé" ("I agree, sign me up"), irás al panel de control de tu recién creado usuario.

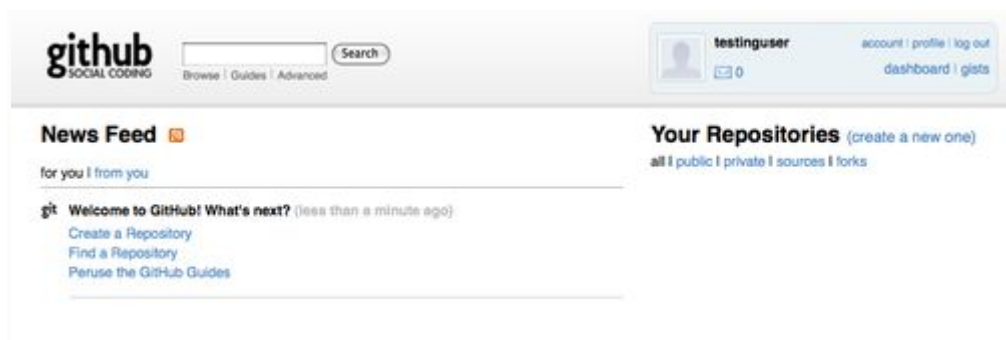
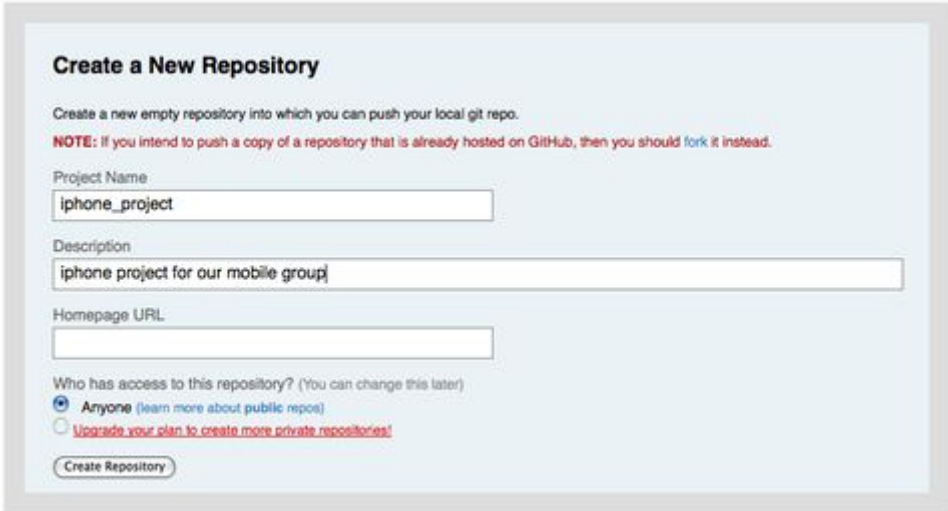


Figura 4-4. El panel de control del usuario GitHub.

A continuación, puedes crear nuevos repositorios.

## Creando un nuevo repositorio

Puedes empezar clicando sobre el enlace "crear uno nuevo" ("create a new one"), en la zona 'Tus repositorios' ('Your Repositories') del panel de control. Irás al formulario de Crear un Nuevo Repositorio (ver Figura 4-5).



The screenshot shows the 'Create a New Repository' form on GitHub. The form has a light blue background and a white border. It contains the following fields and options:

- Create a New Repository** (Section Header)
- Create a new empty repository into which you can push your local git repo.** (Text)
- NOTE:** If you intend to push a copy of a repository that is already hosted on GitHub, then you should [fork it](#) instead. (Text)
- Project Name:** A text input field containing 'iphone\_project'.
- Description:** A text input field containing 'iphone project for our mobile group'.
- Homepage URL:** An empty text input field.
- Who has access to this repository? (You can change this later)** (Text)
- ☒ **Anyone** ([learn more about public repos](#)) (Radio button and text)
- ☐ [Upgrade your plan to create more private repositories!](#) (Radio button and text)
- Create Repository** (Button)

Figura 4-5. Creando un nuevo repositorio en GitHub.

Es suficiente con dar un nombre al proyecto, pero también puedes añadirle una descripción. Cuando lo hayas escrito, clicas sobre el botón "Crear Repositorio" ("Create Repository"). Y ya tienes un nuevo repositorio en GitHub (ver Figura 4-6)



The screenshot shows the GitHub repository page for 'testinguser / iphone\_project'. The page has a light gray background and a white border. It contains the following information:

- github** SOCIAL CODING (Logo and text)
- testinguser** (Username)
- account | profile | log out** (Links)
- dashboard | gists** (Links)
- testinguser / iphone\_project** (Repository name)
- edit** (Button)
- unwatch** (Button)
- Description:** iphone project for our mobile group [edit](#) (Text and link)
- Homepage:** [Click to edit](#) [edit](#) (Text and link)
- Public Clone URL:** [git:github.com/testinguser/iphone\\_project.git](https://github.com/testinguser/iphone_project.git) (Text and icon)
- Your Clone URL:** [git@github.com:testinguser/iphone\\_project.git](https://github.com/testinguser/iphone_project.git) (Text and icon)

Figura 4-6. Información de cabecera de un proyecto GitHub.

Como aún no tienes código, GitHub mostrará instrucciones sobre cómo iniciar un nuevo proyecto, cómo enviar (push) un proyecto Git preexistente, o cómo importar un proyecto desde un repositorio público Subversion (ver Figura 4-7).



Figura 4-7. Instrucciones para un nuevo repositorio.

Estas instrucciones son similares a las que ya hemos visto. Para inicializar un proyecto, no siendo aún un proyecto Git, sueles utilizar:

```
$ git init
$ git add .
$ git commit -m 'initial commit'
```

Una vez tengas un repositorio local Git, añadele el sitio GitHub como un remoto y envía (push) allí tu rama principal:

```
$ git remote add origin git@github.com:testinguser/iphone_project.git
$ git push origin master
```

Así, tu proyecto estará alojado en GitHub; y podrás dar su URL a cualquiera con quien desees compartirlo. En este ejemplo, la URL es [http://github.com/testinguser/iphone\\_project](http://github.com/testinguser/iphone_project). En la página de cabecera de cada uno de tus proyectos, podrás ver dos URLs (ver Figura 4-8).

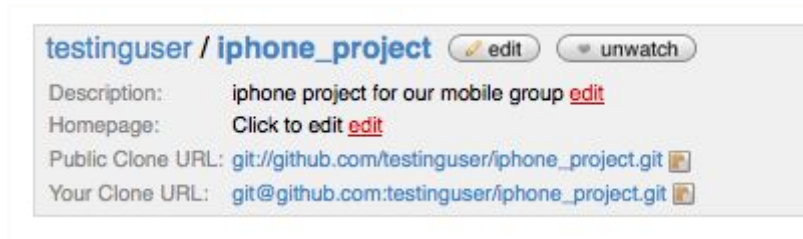


Figura 4-8. Cabecera de proyecto, con una URL pública y otra URL privada.

El enlace "Public Clone URL", es un enlace público, de solo lectura; a través del cual cualquiera puede clonar el proyecto. Puedes comunicar libremente ese URL o puedes publicarlo en tu sitio web o en cualquier otro medio que desees.

El enlace "Your Clone URL", es un enlace de lectura/escritura basado en SSH; a través del cual puedes leer y escribir, pero solo si te conectas con la clave SSH privada correspondiente a la clave pública que has cargado para tu usuario. Cuando otros usuarios visiten la página del proyecto, no verán esta segunda URL --solo verán la URL pública--.

## Importación desde Subversion

Si tienes un proyecto público Subversion que desees pasar a Git, GitHub suele poder realizar la importación. All fondo de la página de instrucciones, tienes un enlace "Subversion import". Si clicas sobre dicho enlace, verás un formulario con información sobre el proceso de importación y un cuadro de texto donde puedes pegar la URL de tu proyecto Subversion (ver Figura 4-9).



The screenshot shows the 'Import a Subversion Repository' page on GitHub. It has a light blue header with the title 'Import a Subversion Repository'. Below the title is a section 'Read Before Proceeding' with three bullet points: 1. The import process could take as little as 5 minutes to as long as 5 days depending on the size of your repository. This has everything to do with how slow subversion is, but we're working on speeding up the process. 2. If your subversion repository contains a non-standard directory structure, this import process will probably not work for you. Check out our [Guide](#) for running the import yourself. 3. This service currently only supports public subversion repositories. Below this is a form with 'Project Name' set to 'iphone\_project' and 'SVN Repository URL' with an empty text box and a help icon. At the bottom is a button labeled 'Step 1: Import Authors'.

Figura 4-9. El interface de importación desde Subversion.

Si tu proyecto es muy grande, no-estandar o privado, es muy posible que no se pueda importar. En el capítulo 7, aprenderás cómo realizar importaciones manuales de proyectos complejos.

## Añadiendo colaboradores

Vamos a añadir al resto del equipo. Si tanto John, como Josie, como Jessica, todos ellos registran sus respectivas cuentas en GitHub. Y deseas darles acceso de escritura a tu repositorio. Puedes incluirlos en tu proyecto como colaboradores. De esta forma, funcionarán los envíos (push) desde sus respectivas claves públicas.

Has de hacer clic sobre el botón "edit" en la cabecera del proyecto o en la pestaña Admin de la parte superior del proyecto; yendo así a la página de administración del proyecto GitHub.

The screenshot shows the GitHub repository administration page for 'testinguser / iphone\_project'. The header shows the repository name and a '1' icon. The main content area lists repository details: Description: 'iphone project for our mobile group' (with an 'edit' link), Homepage: 'Click to edit' (with an 'edit' link), Public Clone URL: 'git://github.com/testinguser/iphone\_project.git', Your Clone URL: 'git@github.com:testinguser/iphone\_project.git', GitHub Page: 'Generate Your Project Page', Pull Requests: 'Add auto-responder', RubyGem: 'Inactive' (with an 'edit' link), and Donations: 'Inactive' (with an 'edit' link). Below this are two sidebars: 'Privacy' on the left, stating 'This repo is viewable by everyone. If you upgrade your plan you can make this repository private!', and 'Repository Collaborators' on the right, with a link 'Add another collaborator'.

Figura 4-10. Página de administración GitHub.

Para dar acceso de escritura a otro usuario, clicas sobre el enlace "Add another collaborator". Aparecerá un cuadro de texto, donde podrás teclear un nombre. Según tecleas, aparecerá un cuadro de ayuda, mostrando posibles nombres de usuario que encajen con lo tecleado. Cuando localices al usuario deseado, clicas sobre el botón "Add" para añadirlo como colaborador en tu proyecto (ver Figura 4-11).

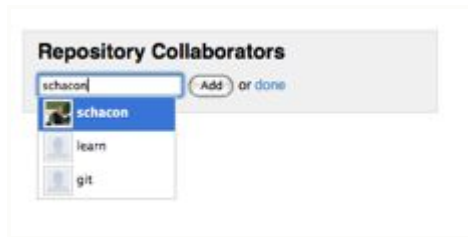


Figura 4-11. Añadiendo un colaborador a tu proyecto.

Cuando termines de añadir colaboradores, podrás ver a todos ellos en la lista "Repository Collaborators" (ver Figura 4-12).

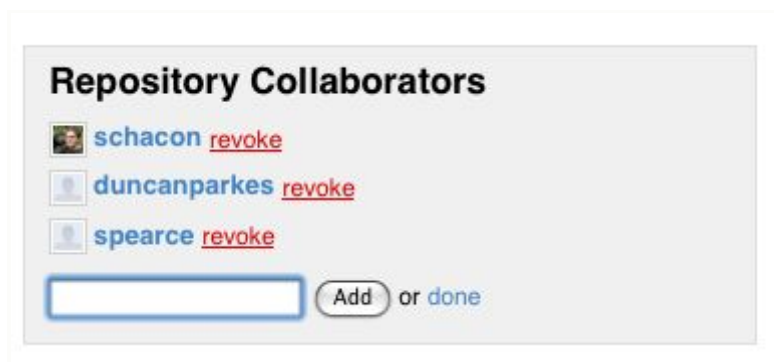


Figura 4-12. Lista de colaboradores en tu proyecto.

Si deseas revocar el acceso a alguno de ellos, puedes clicar sobre el enlace "revoke", y sus permisos de envío (push) serán revocados. En proyectos futuros, podras incluir también a tu grupo de colaboradores copiando los permisos desde otro proyecto ya existente.

## **Tu proyecto**

Una vez hayas enviado (push) tu proyecto, o lo hayas importado desde Subversion, tendrás una página principal de proyecto tal como:

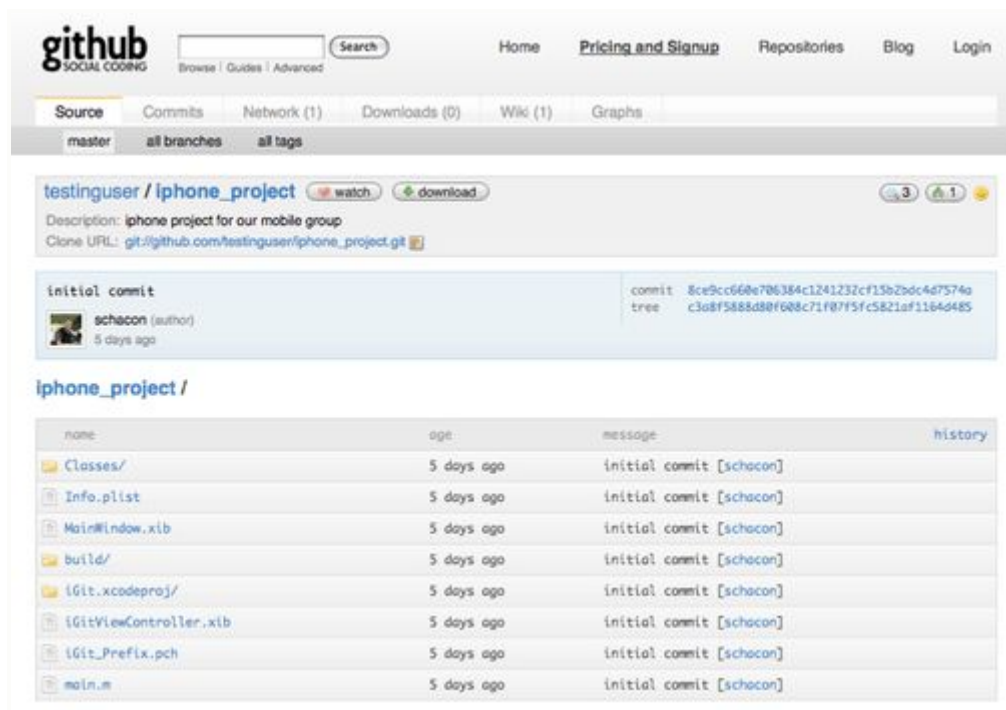


Figura 4-13. Una página principal de proyecto GitHub.

Cuando la gente visite tu proyecto, verá esta página. Tiene pestañas que llevan a distintos aspectos del proyecto. La pestaña "Commits" muestra una lista de confirmaciones de cambio, en orden cronológico inverso, de forma similar a la salida del comando 'git log'. La pestaña "Network" muestra una lista de toda la gente que ha bifurcado (forked) tu proyecto y ha contribuido a él. La pestaña "Downloads" permite cargar binarios del proyecto y enlaza con tarballs o versiones comprimidas de cualquier punto marcado (tagged) en tu proyecto. La pestaña "Wiki" enlaza con un espacio wiki donde puedes escribir documentación o cualquier otra información relevante sobre tu proyecto. La pestaña "Graphs" muestra diversas visualizaciones sobre contribuciones y estadísticas de tu proyecto. La pestaña principal "Source" en la que aterrizas cuando llegas al proyecto, muestra un listado de la carpeta principal; y muestra también el contenido del archivo README, si tienes uno en ella. Esta pestaña muestra también un cuadro con información sobre la última confirmación de cambio (commit) realizada en el proyecto.

## Bifurcando proyectos

Si deseas contribuir a un proyecto ya existente, en el que no tengas permisos de envío (push). GitHub recomienda bifurcar el proyecto. Cuando aterrizas en la página de un proyecto que te parece interesante y con el que deseas trastear un poco, puedes clicar sobre el botón "fork" de la

cabecera del proyecto; de tal forma que GitHub haga una copia del proyecto a tu cuenta de usuario y puedas así enviar (push) cambios sobre él.

De esta forma, los proyectos no han de preocuparse de añadir usuarios como colaboradores para darles acceso de envío (push). La gente puede bifurcar (fork) un proyecto y enviar (push) sobre su propia copia. El gestor del proyecto principal, puede recuperar (pull) esos cambios añadiendo las copias como remotos y fusionando (merge) el trabajo en ellas contenido.

Para bifurcar un proyecto, visita su página (en el ejemplo, `mojombo/chronic`) y clicas sobre el botón "fork" de su cabecera (ver Figura 4-14)



Figura 4-14. Obtener una copia sobre la que escribir, clicando sobre el botón "fork" de un repositorio.

Tras unos segundos, serás redirigido a la página del nuevo proyecto; y en ella se verá que este proyecto es una bifurcación (fork) de otro existente (ver Figura 4-15).

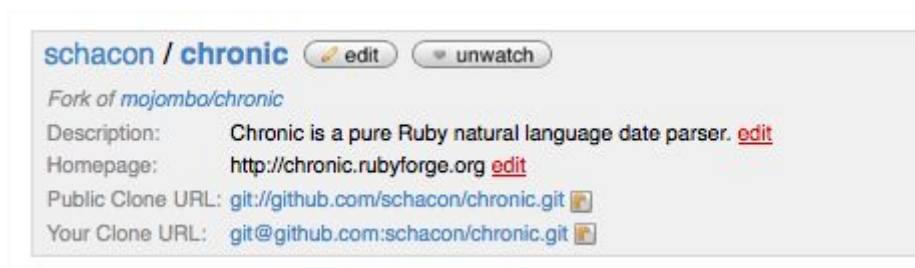


Figura 4-15. Tu bifurcación (fork) de un proyecto.

## Resumen de GitHub

Esto es todo lo que vamos a ver aquí sobre GitHub, pero merece la pena destacar lo rápido que puedes hacer todo esto. Puedes crear una cuenta, añadir un nuevo proyecto y contribuir a él en cuestión de minutos. Si tu proyecto es de código abierto, puedes tener también una amplia comunidad de desarrolladores que podrán ver tu proyecto, bifurcarlo (fork) y ayudar contribuyendo a él. Y, por último, comentar que esta puede ser una buena manera de iniciarte y comenzar rápidamente a trabajar con Git.

# Chapter 5

## Git en entornos distribuidos

Ahora que ya tienes un repositorio Git, configurado como punto de trabajo para compartir código entre desarrolladores. Y ahora que ya conoces los comandos básicos de Git para flujos de trabajo locales. Puedes echar un vistazo a algunos de los flujos de trabajo distribuidos que Git permite.

En este capítulo, verás cómo trabajar con Git en un entorno distribuido, bien como colaborador o bien como integrador. Es decir, aprenderás cómo contribuir adecuadamente a un proyecto; de la forma más efectiva posible, tanto para tí, como para quien gestione el proyecto. Y aprenderás también a gestionar proyectos en los que colaboren multiples desarrolladores.

### 5.1 Git en entornos distribuidos - Flujos de trabajo distribuidos

#### Flujos de trabajo distribuidos

Al contrario de otros Sistemas Centralizados de Control de Versiones, (CVCSs, Centralized Version Control Systems), la naturaleza distribuida de Git permite mucha más flexibilidad en la manera de colaborar en proyectos. En los sistemas centralizados, cada desarrollador es un nodo de trabajo; trabajando todos ellos, en pie de igualdad, sobre un mismo repositorio central. En Git, en cambio, cada desarrollador es potencialmente tanto un nodo como un repositorio --es decir, cada desarrollador puede tanto contribuir a otros repositorios, como servir de repositorio público sobre el que otros desarrolladores pueden basar su trabajo y contribuir a él--. Esto abre un enorme rango de posibles formas de trabajo en tu proyecto y/o en tu equipo. Aquí vamos a revisar algunos de los paradigmas más comunes diseñados para sacar ventaja a esta gran flexibilidad. Vamos a repasar las fortalezas y posibles debilidades de cada paradigma. En tu trabajo, podrás elegir solo uno concreto, o podrás mezclar escogiendo funcionalidades concretas de cada uno.

#### Flujo de trabajo centralizado

En los sistemas centralizados, tenemos una única forma de trabajar. Un repositorio o punto central guarda el código fuente; y todo el mundo sincroniza su trabajo con él. Unos cuantos desarrolladores son nodos de trabajo --consumidores de dicho repositorio-- y se sincronizan con dicho punto central. (ver Figura 5-1).

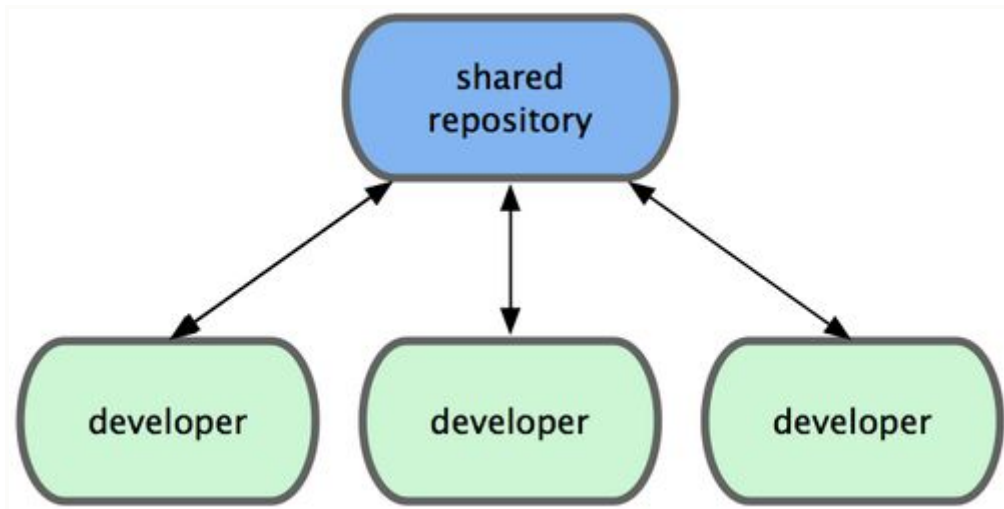


Figura 5-1. Flujo de trabajo centralizado.

Esto significa que, si dos desarrolladores clonan desde el punto central, y ambos hacen cambios; tan solo el primero de ellos en enviar sus cambios de vuelta lo podrá hacer limpiamente. El segundo desarrollador deberá fusionar previamente su trabajo con el del primero, antes de enviarlo, para evitar el sobrescribir los cambios del primero. Este concepto es también válido en Git, tanto como en Subversion (o cualquier otro CVCS), y puede ser perfectamente utilizado en Git.

Si tienes un equipo pequeño o te sientes confortable con un flujo de trabajo centralizado, puedes continuar usando esa forma de trabajo con Git. Solo necesitas disponer un repositorio único, y dar acceso en envío (push) a todo tu equipo. Git se encargará de evitar el que se sobrescriban unos a otros. Si uno de los desarrolladores clona, hace cambios y luego intenta enviarlos; y otro desarrollador ha enviado otros cambios durante ese tiempo; el servidor rechazará los cambios del segundo desarrollador. El sistema le avisará de que está intentando enviar (push) cambios no directos (non-fast-forward changes), y de que no podrá hacerlo hasta que recupere (fetch) y fusione (merge) los cambios preexistentes. Esta forma de trabajar es atractiva para mucha gente, por ser el paradigma con el que están familiarizados y se sienten confortables.

### **Flujo de trabajo del Gestor-de-Integraciones**

Al permitir multiples repositorios remotos, en Git es posible tener un flujo de trabajo donde cada desarrollador tenga acceso de escritura a su propio repositorio público y acceso de lectura a los repositorios de todos los demás. Habitualmente, este escenario suele incluir un repositorio canónico, representante "oficial" del proyecto. Para contribuir en este tipo de proyecto, crearás tu propio clón público del mismo y enviarás (push) tus cambios a este. Después, enviarás una petición a la persona gestora del proyecto principal, para que recupere y consolide (pull) en él tus cambios. Ella podrá añadir tu repositorio como un remoto, chequear tus cambios localmente, fusionarlos (merge) con su rama y enviarlos (push) de vuelta a su repositorio. El proceso funciona de la siguiente manera (ver Figura 5-2):

1. La persona gestora del proyecto envia (push) a su repositorio público (repositorio principal).
2. Una persona que desea contribuir, clona dicho repositorio y hace algunos cambios.
3. La persona colaboradora envia (push) a su propia copia pública.
4. Esta persona colaboradora envia a la gestora un correo electronico solicitándole recupere e integre los cambios.
5. La gestora añade como remoto el repositorio de la colaboradora y fusiona (merge) los cambios localmente.
6. La gestora envia (push) los cambios fusionados al repositorio principal.

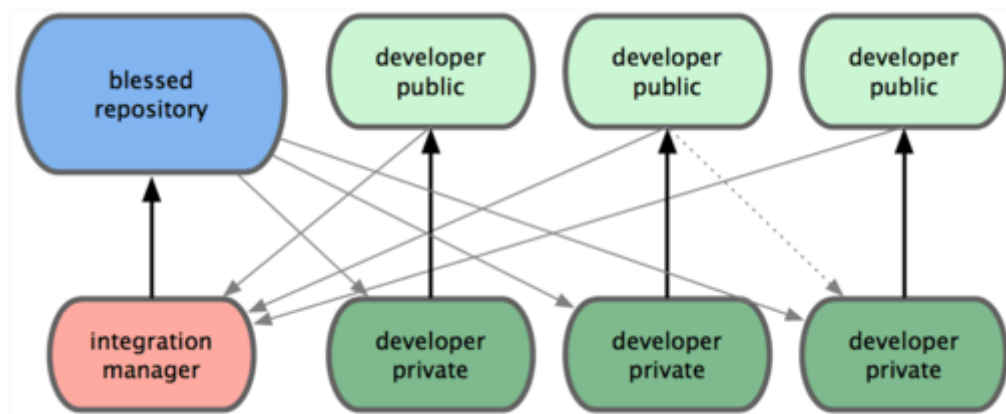


Figura 5-2. Flujo de trabajo Gestor-de-Integración.

Esta es una forma de trabajo muy común en sitios tales como GitHub, donde es sencillo bifurcar (fork) un proyecto y enviar tus cambios a tu copia, donde cualquiera puede verlos. La principal ventaja de esta forma de trabajar es que puedes continuar trabajando, y la persona gestora del repositorio principal podrá recuperar (pull) tus cambios en cualquier momento. Las personas



colaboradoras no tienen por qué esperar a que sus cambios sean incorporados al proyecto, --cada cual puede trabajar a su propio ritmo--.

### Flujo de trabajo con Dictador y Tenientes

Es una variante del flujo de trabajo con múltiples repositorios. Se utiliza generalmente en proyectos muy grandes, con cientos de colaboradores. Un ejemplo muy conocido es el del kernel de Linux. Unos gestores de integración se encargan de partes concretas del repositorio; y se denominan tenientes. Todos los tenientes rinden cuentas a un gestor de integración; conocido como el dictador benevolente. El repositorio del dictador benevolente es el repositorio de referencia, del que recuperan (pull) todos los colaboradores. El proceso funciona como sigue (ver Figura 5-3):

1. Los desarrolladores habituales trabajan cada uno en su rama puntual y reorganizan (rebase) su trabajo sobre la rama master. La rama master es la del dictador benevolente.
2. Los tenientes fusionan (merge) las ramas puntuales de los desarrolladores sobre su propia rama master.
3. El dictador fusiona las ramas master de los tenientes en su propia rama master.
4. El dictador envía (push) su rama master al repositorio de referencia, para permitir que los desarrolladores reorganicen (rebase) desde ella.

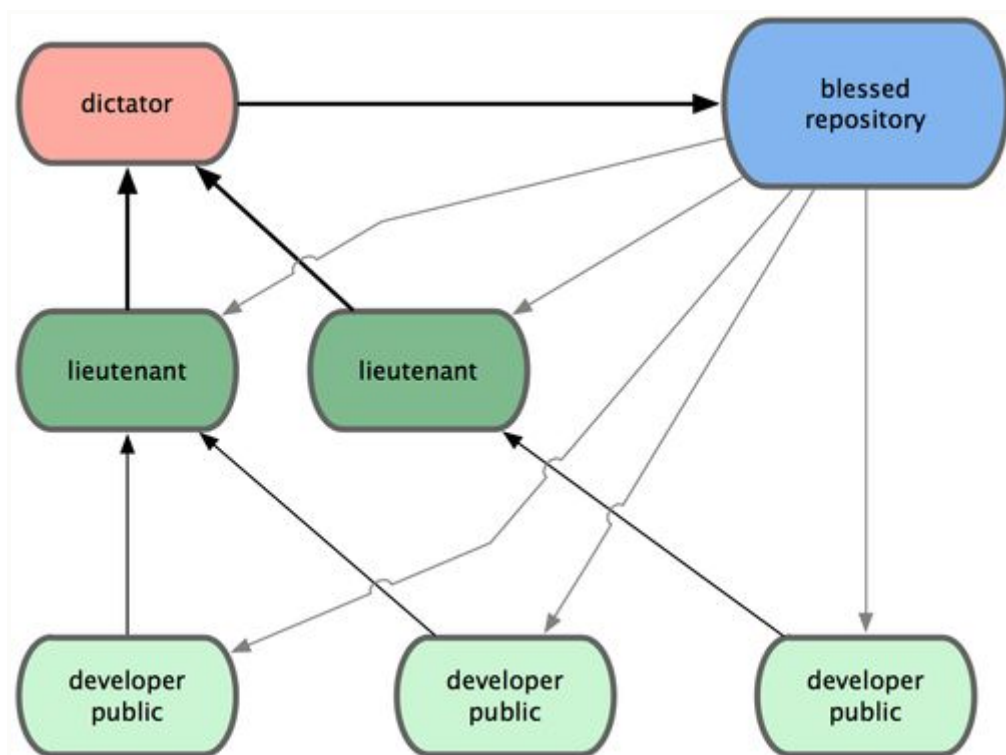


Figura 5-3. Flujo de trabajo del dictador benevolente.

Esta manera de trabajar no es muy habitual, pero es muy útil en proyectos muy grandes o en organizaciones fuertemente jerarquizadas. Permite al líder o a la líder del proyecto (el/la dictador/a) delegar gran parte del trabajo; recolectando el fruto de múltiples puntos de trabajo antes de integrarlo en el proyecto.

Hemos visto algunos de los flujos de trabajo más comunes permitidos por un sistema distribuido como Git. Pero seguro que habrás comenzado a vislumbrar múltiples variaciones que puedan encajar con tu particular forma de trabajar. Espero que a estas alturas estés en condiciones de reconocer la combinación de flujos de trabajo que puede ser útil. Vamos a ver algunos ejemplos más específicos, ilustrativos de los roles principales que se presentan en las distintas maneras de trabajar.

## 5.2 Git en entornos distribuidos - Contribuyendo a un proyecto

### Contribuyendo a un proyecto

En estos momentos conoces las diferentes formas de trabajar, y tienes ya un generoso conocimiento de los fundamentos de Git. En esta sección, aprenderás acerca de las formas más habituales de contribuir a un proyecto.

El mayor problema al intentar describir este proceso es el gran número de variaciones que se pueden presentar. Por la gran flexibilidad de Git, la gente lo suele utilizar de múltiples maneras; siendo problemático intentar describir la forma en que deberías contribuir a un proyecto --cada proyecto tiene sus peculiaridades--. Algunas de las variables a considerar son: la cantidad de colaboradores activos, la forma de trabajo escogida, el nivel de acceso que tengas, y, posiblemente, el sistema de colaboración externa implantado.

La primera variable es el número de colaboradores activos. ¿Cuántos usuarios están enviando activamente código a este proyecto?, y ¿con qué frecuencia?. En muchas ocasiones, tendrás dos o tres desarrolladores, con tan solo unas pocas confirmaciones de cambios (commits) diarias; e incluso menos en algunos proyectos durmientes. En proyectos o empresas verdaderamente grandes puedes tener cientos de desarrolladores, con docenas o incluso cientos de parches

llegando cada día. Esto es importante, ya que cuantos más desarrolladores haya, mayores serán los problemas para asegurarte de que tu código se integre limpiamente. Los cambios que envíes pueden quedar obsoletos o severamente afectados por otros trabajos que han sido integrados previamente mientras tú estabas trabajando o mientras tus cambios aguardaban a ser aprobados para su integración. ¿Cómo puedes mantener consistentemente actualizado tu código, y asegurarte así de que tus parches son válidos?

La segunda variable es la forma de trabajo que se utilice para el proyecto. ¿Es centralizado, con iguales derechos de acceso en escritura para cada desarrollador?. ¿Tiene un gestor de integraciones que comprueba todos los parches?. ¿Se revisan y aprueban los parches entre los propios desarrolladores?. ¿Participas en ese proceso de aprobación?. ¿Existe un sistema de tenientes, a los que has de enviar tu trabajo en primer lugar?.

La tercera variable es el nivel de acceso que tengas. La forma de trabajar y de contribuir a un proyecto es totalmente diferente dependiendo de si tienes o no acceso de escritura al proyecto. Si no tienes acceso de escritura, ¿cuál es el sistema preferido del proyecto para aceptar contribuciones?. Es más, ¿tiene un sistema para ello?. ¿Con cuánto trabajo contribuyes?. ¿Con qué frecuencia?.

Todas estas preguntas afectan a la forma efectiva de contribuir a un proyecto, y a la forma de trabajo que prefieras o esté disponible para tí. Vamos a cubrir ciertos aspectos de todo esto, en una serie de casos de uso; desde los más sencillos hasta los más complejos. A partir de dichos ejemplos, tendrías que ser capaz de construir la forma de trabajo específica que necesites para cada caso.

## **Reglas para las confirmaciones de cambios (commits)**

Antes de comenzar a revisar casos de uso específicos, vamos a dar una pincelada sobre los mensajes en las confirmaciones de cambios (commits). Disponer unas reglas claras para crear confirmaciones de cambios, y seguirlas fielmente, facilita enormemente tanto el trabajo con Git y como la colaboración con otras personas. El propio proyecto Git suministra un documento con un gran número de buenas sugerencias sobre la creación de confirmaciones de cambio destinadas a enviar parches --puedes leerlo en el código fuente de Git, en el archivo 'Documentation/SubmittingPatches'--.

En primer lugar, no querrás enviar ningún error de espaciado. Git nos permite buscarlos fácilmente. Previamente a realizar una confirmación de cambios, lanzar el comando 'git diff

--check' para identificar posibles errores de espaciado. Aquí van algunos ejemplos, en los que hemos sustituido las marcas rojas por 'X's:

```
$ git diff --check
lib/simplegit.rb:5: trailing whitespace.
+ @git_dir = File.expand_path(git_dir)XX
lib/simplegit.rb:7: trailing whitespace.
+ XXXXXXXXXXXXX
lib/simplegit.rb:26: trailing whitespace.
+ def command(git_cmd)XXXX
```

Lanzando este comando antes de confirmar cambios, puedes estar seguro de si vas o no a incluir problemas de espaciado que puedan molestar a otros desarrolladores.

En segundo lugar, intentar hacer de cada confirmación (commit) un grupo lógico e independiente de cambios. Siempre que te sea posible, intenta hacer digeribles tus cambios --no estés trabajando todo el fin de semana, sobre cuatro o cinco asuntos diferentes, y luego confirmes todo junto el lunes--. Aunque no hagas ninguna confirmación durante el fin de semana, el lunes puedes utilizar el área de preparación (staging area) para ir dividiendo tu trabajo y hacer una confirmación de cambios (commit) separada para cada asunto; con un mensaje adecuado para cada una de ellas. Si algunos de los cambios modifican un mismo archivo, utiliza el comando 'git add --patch' para almacenar parcialmente los archivos (tal y como se verá detalladamente en el Capítulo 6). El estado del proyecto al final de cada rama será idéntico si haces una sola confirmación o si haces cinco, en tanto en cuanto todos los cambios estén confirmados en un determinado momento. Por consiguiente, intenta facilitar las cosas a tus compañeros y compañeras desarrolladores cuando vayan a revisar tus cambios. Además, esta manera de trabajar facilitará la integración o el descarte individual de alguno de los cambios, en caso de ser necesario. El Capítulo 6 contiene un gran número de trucos para reescribir el historial e ir preparando archivos interactivamente --utilízalos para ayudarte a crear un claro y comprensible historial--.

Y, por último, prestar atención al mensaje de confirmación. Si nos acostumbramos a poner siempre mensajes de calidad en las confirmaciones de cambios, facilitaremos en gran medida el trabajo y la colaboración con Git. Como regla general, tus mensajes han de comenzar con una línea, de no más de 50 caracteres, donde se resuma el grupo de cambios; seguida de una línea en blanco; y seguida de una detallada explicación en las líneas siguientes. El proyecto Git obliga a incluir una explicación detallada; incluyendo tus motivaciones para los cambios realizados; y comentarios sobre las diferencias, tras su implementación, respecto del comportamiento anterior

del programa. Esta recomendación es una buena regla a seguir. Es también buena idea redactar los mensajes utilizando el imperativo, en tiempo presente. Es decir, dá órdenes. Por ejemplo, en vez de escribir "He añadido comprobaciones para" o "Añadiendo comprobaciones para", utilizar la frase "Añadir comprobaciones para". Como plantilla de referencia, podemos utilizar la que escribió Tim Pope en [tpope.net](http://tpope.net):

Un resumen de cambios, corto (50 caracteres o menos).

Seguido de un texto más detallado, si es necesario. Limitando las líneas a 72 caracteres mas o menos. En algunos contextos, la primera línea se tratará como si fuera el asunto de un correo electrónico y el resto del texto como si fuera el cuerpo. La línea en blanco separando el resumen del cuerpo es crítica (a no ser que se omita totalmente el cuerpo); algunas herramientas como 'rebase' pueden tener problemas si no los separas adecuadamente.

Los siguientes párrafos van tras la línea en blanco.

- Los puntos con bolo también están permitidos.
- Habitualmente se emplea un guión o un asterisco como bolo, seguido de un espacio, con líneas en blanco intermedias; pero las convenciones pueden variar.

Si todas tus confirmaciones de cambio (commit) llevan mensajes de este estilo, facilitarás las cosas tanto para tí como para las personas que trabajen contigo. El proyecto Git tiene los mensajes adecuadamente formateados. Te animo a lanzar el comando 'git log --no-merges' en dicho proyecto, para que veas la pinta que tiene un historial de proyecto limpio y claro.

En los ejemplos siguientes, y a lo largo de todo este libro, por razones de brevedad no formatearé correctamente los mensajes; sino que emplearé la opción '-m' en los comandos 'git commit'. Observa mis palabras, no mis hechos.

## **Pequeño Grupo Privado**

Lo más simple que te puedes encontrar es un proyecto privado con uno o dos desarrolladores. Por privado, me refiero a código propietario --no disponible para ser leído por el mundo exterior--. Tanto tu como el resto del equipo teneis acceso de envío (push) al repositorio.

En un entorno como este, puedes seguir un flujo de trabajo similar al que adoptarías usando Subversion o algún otro sistema centralizado. Sigues disfrutando de ventajas tales como las confirmaciones offline y la mayor simplicidad en las ramificaciones/fusiones. Pero, en el fondo, la

forma de trabajar será bastante similar; la mayor diferencia radica en que las fusiones (merge) se hacen en el lado cliente en lugar de en el servidor. Vamos a ver como actuarían dos desarrolladores trabajando conjuntamente en un repositorio compartido. El primero de ellos, John, clona el repositorio, hace algunos cambios y los confirma localmente: (en estos ejemplos estoy sustituyendo los mensajes del protocolo con '...' , para acortarlos)

```
# John's Machine
$ git clone john@github: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(-)
```

La segunda desarrolladora, Jessica, hace lo mismo: clona el repositorio y confirma algunos cambios:

```
# Jessica's Machine
$ git clone jessica@github: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(-)
```

Tras esto, Jessica envía (push) su trabajo al servidor:

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

John intenta enviar también sus cambios:

```
# 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 enviar porque Jessica ha enviado previamente. Entender bien esto es especialmente importante, sobre todo si estás acostumbrado a utilizar Subversion; porque habrás notado que ambos desarrolladores han editado archivos distintos. Mientras que Subversion fusiona automáticamente en el servidor cuando los cambios han sido aplicados sobre archivos distintos, en Git has de fusionar (merge) los cambios localmente. John se ve obligado a recuperar (fetch) los cambios de Jessica y a fusionarlos (merge) con los suyos, antes de que se le permita enviar (push):

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

En este punto, el repositorio local de John será algo parecido a la Figura 5-4.

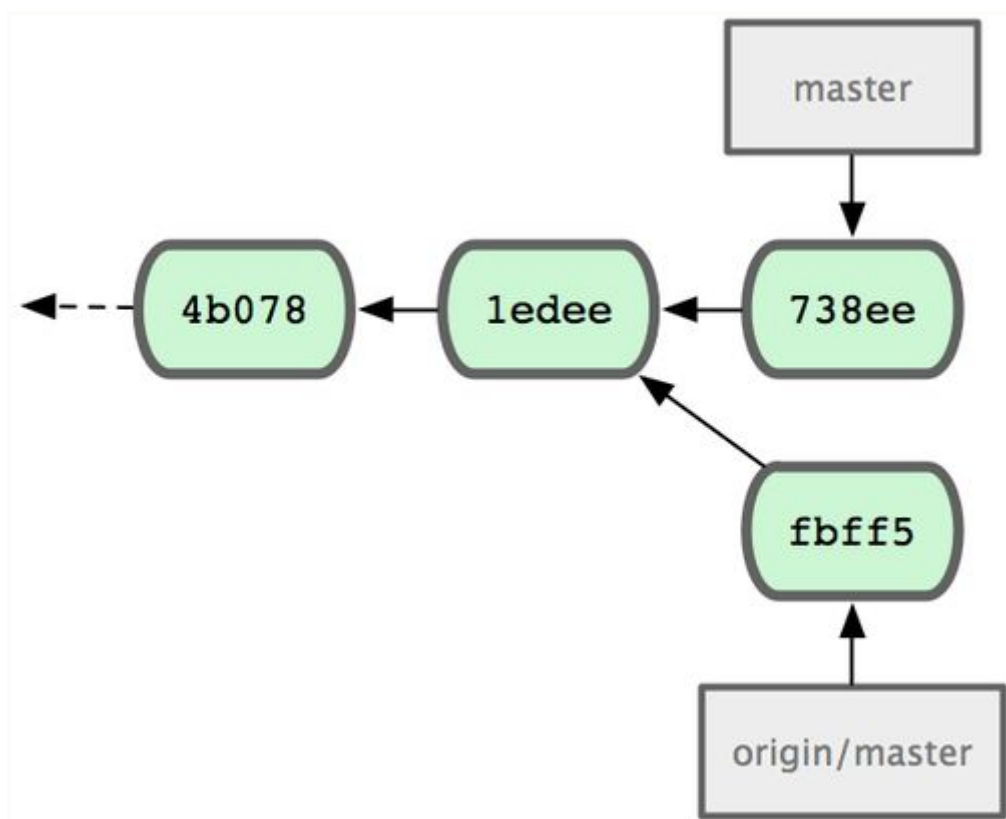


Figura 5-4. El repositorio inicial de John.

John tiene una referencia a los cambios enviados por Jessica, pero ha de fusionarlos en su propio trabajo antes de que se le permita enviar:

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

Si la fusión se realiza sin problemas, el historial de John será algo parecido a la Figura 5-5.

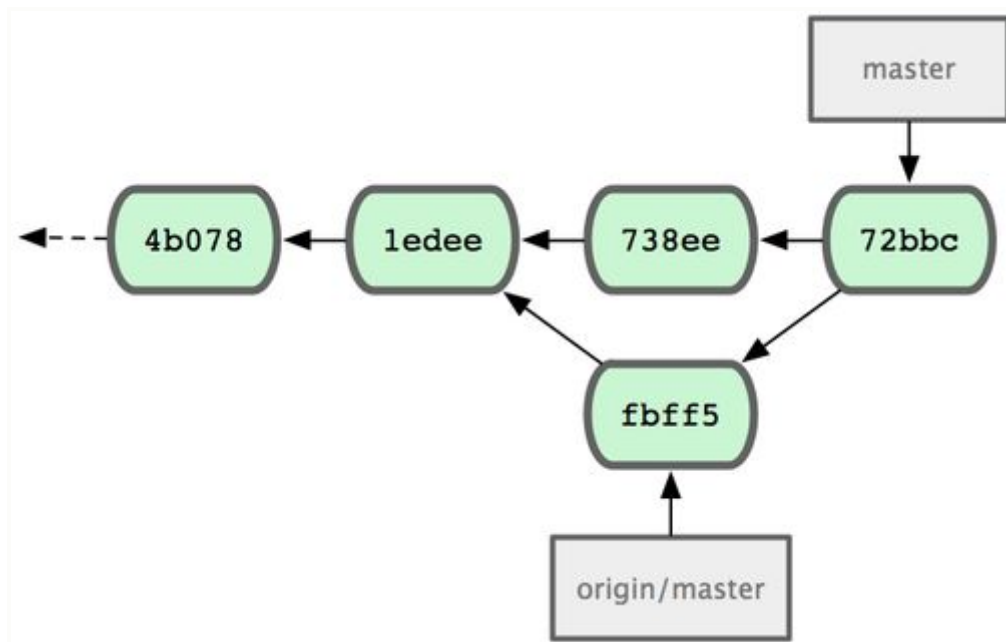


Figura 5-5. El repositorio de John tras fusionar origin/master.

En este momento, John puede comprobar su código para verificar que sigue funcionando correctamente, y luego puede enviar su trabajo al servidor:

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



Finalmente, el historial de John es algo parecido a la Figura 5-6.

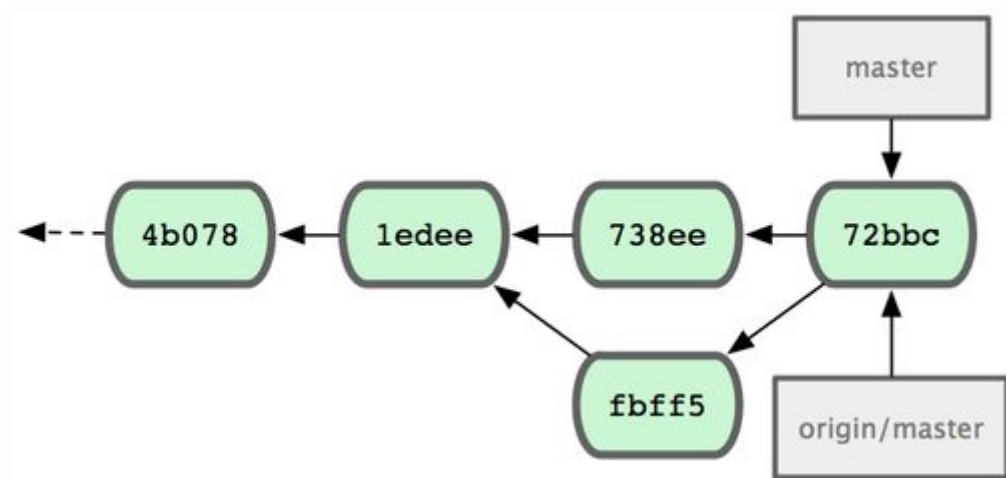


Figura 5-6. El historial de John tras enviar al servidor origen.

Mientras tanto, Jessica ha seguido trabajando en una rama puntual (topic branch). Ha creado una rama puntual denominada 'issue54' y ha realizado tres confirmaciones de cambios (commit) en dicha rama. Como todavía no ha recuperado los cambios de John, su historial es como se muestra en la Figura 5-7.

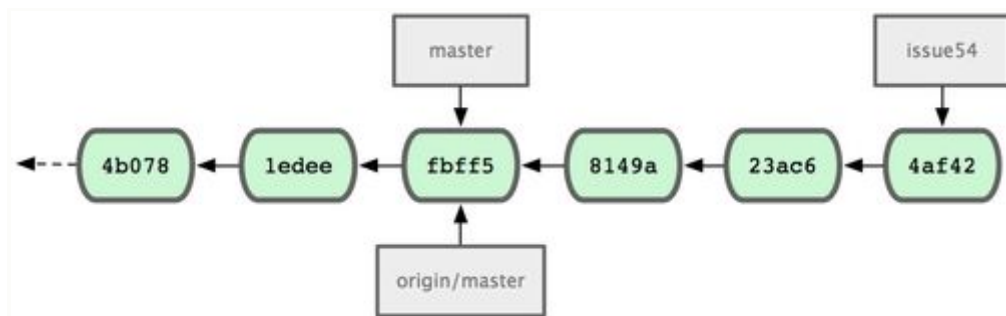


Figura 5-7. Historial inicial de Jessica.

Jessica desea sincronizarse con John, para lo cual:

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

Esto recupera el trabajo enviado por John durante el tiempo en que Jessica estaba trabajando. El historial de Jessica es en estos momentos como se muestra en la figura 5-8.

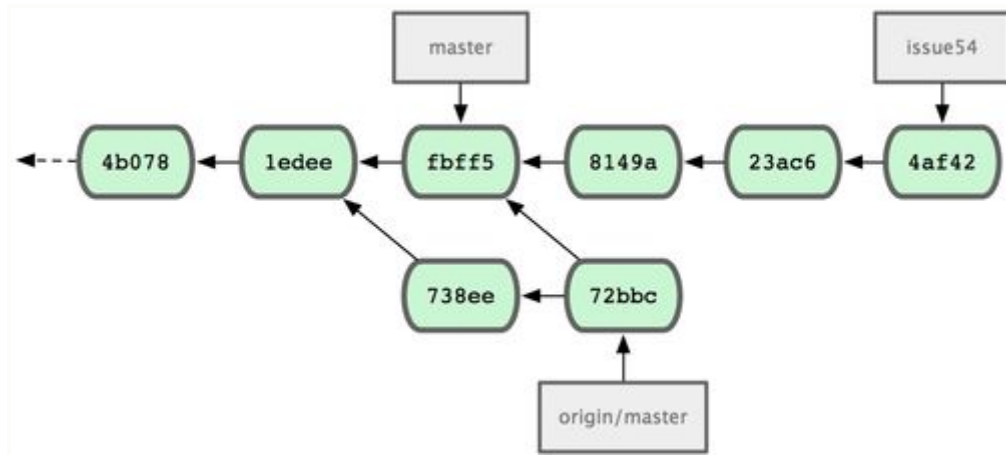


Figura 5-8. El historial de Jessica tras recuperar los cambios de John.

Jessica considera su rama puntual terminada, pero quiere saber lo que debe integrar con su trabajo antes de poder enviarla. Lo comprueba con el comando 'git log':

```
$ git log --no-merges origin/master ^issue54
commit 738ee872852dfaa9d6634e0dea7a324040193016
Author: John Smith <jsmith@example.com>
Date: Fri May 29 16:01:27 2009 -0700
```

removed invalid default value

Ahora, jessica puede fusionar (merge) su trabajo de la rama puntual 'issue54' en su rama 'master', fusionar (merge) el trabajo de John ('origin/master') en su rama 'master', y enviarla de vuelta al servidor. Primero, se posiciona de nuevo en su rama principal para integrar todo su trabajo:

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

Puede fusionar primero tanto 'origin/master' o como 'issue54', ya que ambos están aguas arriba. La situación final (snapshot) será la misma, indistintamente del orden elegido; tan solo el historial variará ligeramente. Elige fusionar primero 'issue54':

```
$ git merge issue54
Updating fbff5bc..4af4298
Fast forward
 README      | 1 +
 lib/simplegit.rb | 6 +++++-
 2 files changed, 6 insertions(+), 1 deletions(-)
```

No hay ningún problema; como puedes observar, es un simple avance rápido (fast-forward). Tras esto, jessica fusiona 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 integra limpiamente, y el historial de Jessica queda como en la Figura 5-9.

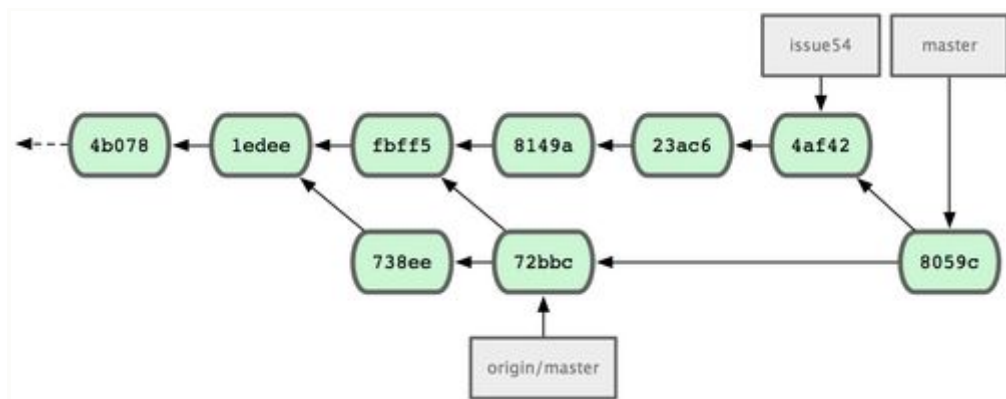


Figura 5-9. El historial de Jessica tras fusionar los cambios de John.

En este punto, la rama 'origin/master' es alcanzable desde la rama 'master' de Jessica, permitiéndole enviar (push) --asumiendo que John no haya enviado nada más durante ese tiempo--:

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

Cada desarrollador ha confirmado algunos cambios y ambos han fusionado sus trabajos correctamente; ver Figura 5-10.

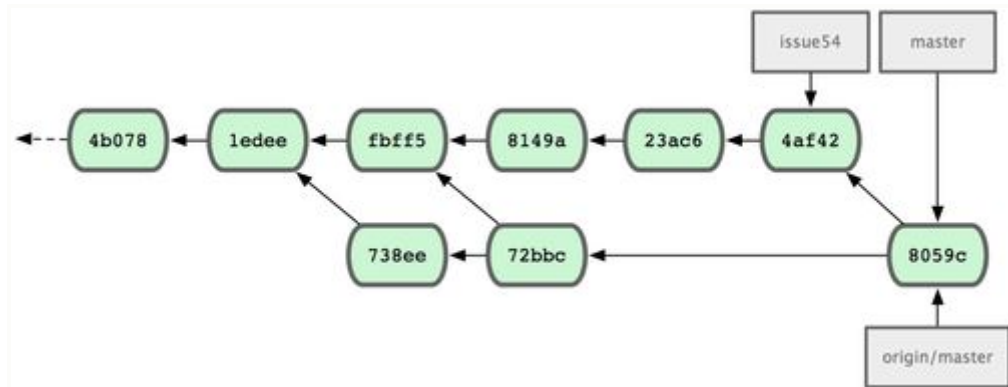


Figura 5-10. El historial de Jessica tras enviar de vuelta todos los cambios al servidor.

Este es uno de los flujos de trabajo más simples. Trabajas un rato, normalmente en una rama puntual de un asunto concreto, y la fusionas con tu rama principal cuando la tienes lista para integrar. Cuando deseas compartir ese trabajo, lo fusionas (merge) en tu propia rama 'master'; luego recuperas (fetch) y fusionas (merge) la rama 'origin/master', por si hubiera cambiado; y finalmente envías (push) la rama 'master' de vuelta al servidor. La secuencia general es algo así como la mostrada en la Figura 5-11.

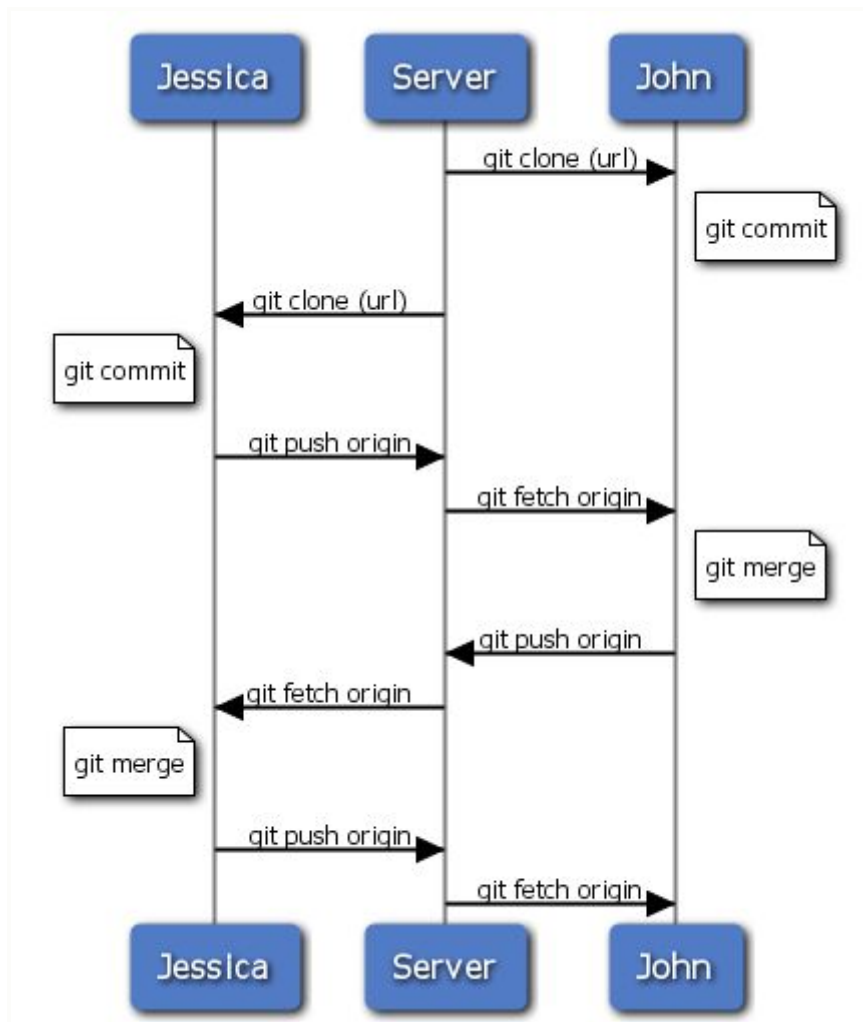


Figura 5-11. Secuencia general de eventos en un flujo de trabajo multidesarrollador simple.

### Grupo Privado Gestionado

En este próximo escenario, vamos a hechar un vistazo al rol de colaborador en un gran grupo privado. Aprenderás cómo trabajar en un entorno donde pequeños grupos colaboran en algunas funcionalidades, y luego todas las aportaciones de esos equipos son integradas por otro grupo.

Supongamos que John y Jessica trabajan conjuntamente en una funcionalidad, mientras que Jessica y Josie trabajan en una segunda funcionalidad. En este caso, la compañía está utilizando un flujo de trabajo del tipo gestor-de-integración, donde el trabajo de algunos grupos individuales es integrado por unos ingenieros concretos; siendo solamente estos últimos quienes pueden actualizar la rama 'master' del repositorio principal. En este escenario, todo el trabajo se realiza en ramas propias de cada grupo y es consolidado por los integradores más tarde.

Vamos a seguir el trabajo de Jessica, a medida que trabaja en sus dos funcionalidades; colaborando en paralelo con dos desarrolladores distintos. Suponiendo que tiene su repositorio ya clonado, ella decide trabajar primero en la funcionalidad A (featureA). Crea una nueva rama para dicha funcionalidad y trabaja en ella:

```
# 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 ese punto, necesita compartir su trabajo con John, por lo que envía (push) al servidor las confirmaciones (commits) en su rama 'featureA'. Como Jessica no tiene acceso de envío a la rama 'master' --solo los integradores lo tienen--, ha de enviar a alguna otra rama para poder colaborar con John:

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

Jessica notifica a John por correo electrónico que ha enviado trabajo a una rama denominada 'featureA' y que puede echarle un vistazo allí. Mientras espera noticias de John, Jessica decide comenzar a trabajar en la funcionalidad B (featureB) con Josie. Para empezar, arranca una nueva rama puntual, basada en la rama 'master' del servidor:

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

Y realiza un par de confirmaciones de cambios (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(-)
```

Quedando su repositorio como se muestra en la Figura 5-12

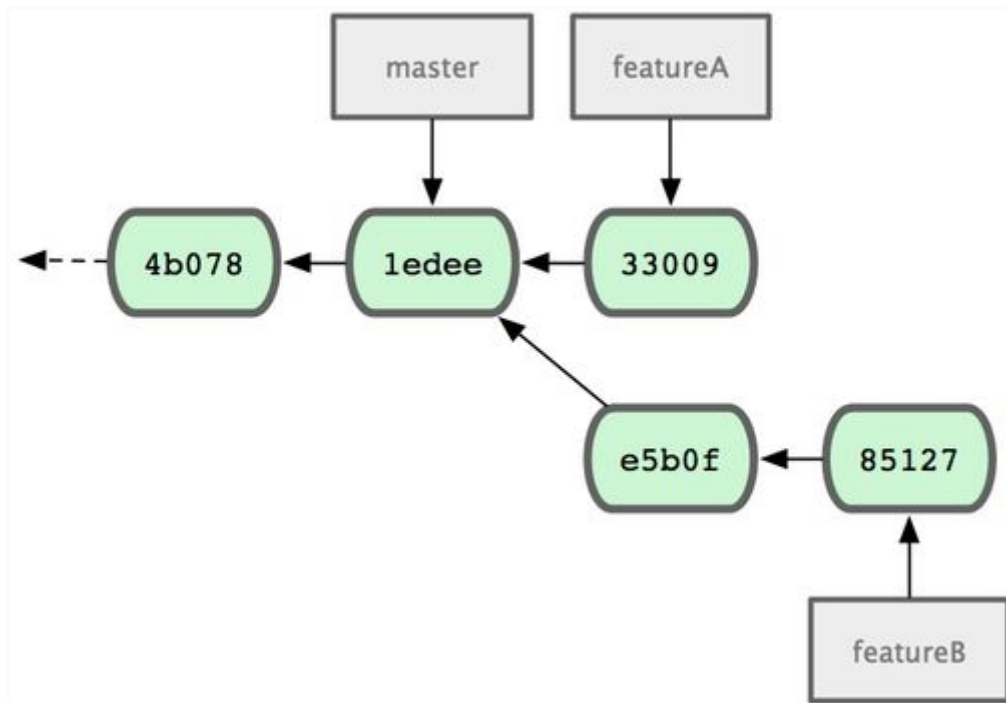


Figura 5-12. Historial inicial de Jessica.

Cuando está preparada para enviar (push) su trabajo, recibe un correo electrónico de Josie de que ha puesto en el servidor una rama denominada 'featureBee', con algo de trabajo. Jessica necesita fusionar (merge) dichos cambios con los suyos antes de poder enviarlos al servidor. Por tanto, recupera (fetch) los cambios de Josie:

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

Y los fusiona con su propio trabajo:

```
$ git merge origin/featureBee
Auto-merging lib/simplegit.rb
Merge made by recursive.
```

```
lib/simplegit.rb | 4 +++++
1 files changed, 4 insertions(+), 0 deletions(-)
```

Pero hay un pequeño problema, necesita enviar el trabajo fusionado en su rama 'featureB' a la rama 'featureBee' del servidor. Puede hacerlo usando el comando 'git push' e indicando específicamente el nombre de la rama local seguida de dos puntos (:) y seguida del nombre de la rama remota:

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

Esto es lo que se denomina un *refspec*. Puedes ver el Capítulo 9 para una discusión más detallada acerca de los *refspecs* de Git y los distintos usos que puedes darles.

A continuación, John envía un correo electrónico a Jessica comentándole que ha enviado algunos cambios a la rama 'featureA' y pidiéndole que los verifique. Ella lanza un 'git fetch' para recuperar dichos cambios:

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

A continuación, puede ver las modificaciones realizadas, lanzando el comando 'git log':

```
$ git log origin/featureA ^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
```

Para terminar, 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 realiza algunos ajustes, los confirma (commit) y los envia (push) de vuelta al servidor:

```
$ git commit -am 'small tweak'  
[featureA ed774b3] small tweak  
1 files changed, 1 insertions(+), 1 deletions(-)  
$ git push origin featureA  
...  
To jessica@github:simplegit.git  
3300904..ed774b3 featureA -> featureA
```

Quedando su historial como se muestra en la Figura 5-13.

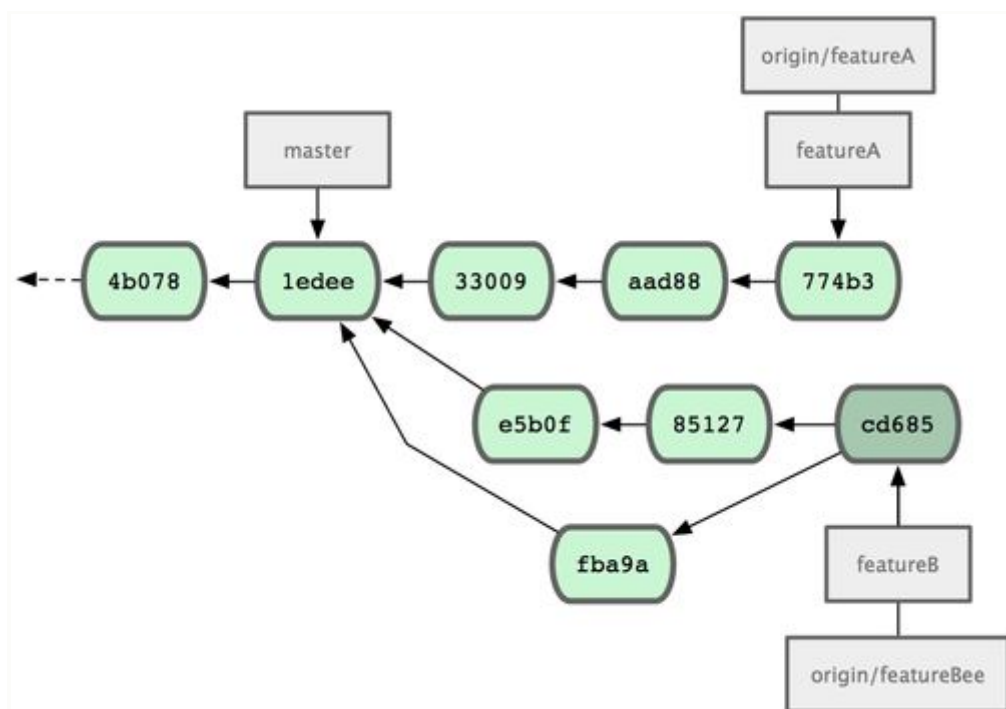


Figura 5-13. El historial de Jessica después de confirmar cambios en una rama puntual.

Jessica, Josie y John informan a los integradores de que las ramas 'featureA' y 'featureBee' del servidor están preparadas para su integración con la línea principal del programa. Después de que dichas ramas sean integradas en la línea principal, una recuperación (fetch) traerá de vuelta

las confirmaciones de cambios de las integraciones (merge commits), dejando un historial como el mostrado en la Figura 5-14.

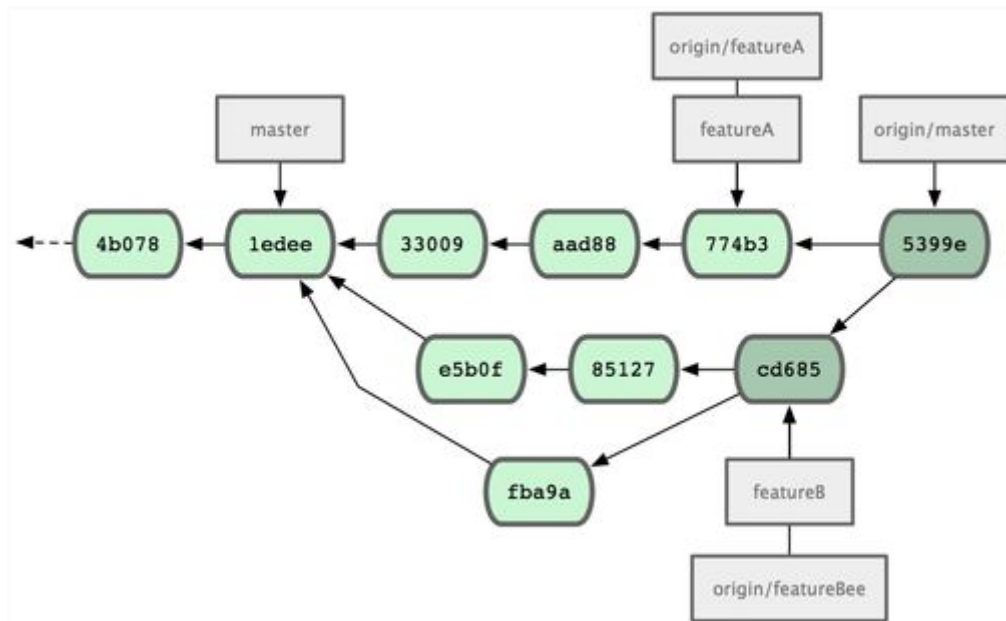


Figura 5-14. El historial de Jessica tras fusionar sus dos ramas puntuales.

Muchos grupos se están pasando a trabajar con Git, debido a su habilidad para mantener múltiples equipos trabajando en paralelo, fusionando posteriormente las diferentes líneas de trabajo. La habilidad para que pequeños subgrupos de un equipo colaboren a través de ramas remotas, sin necesidad de tener en cuenta o de perturbar el equipo completo, es un gran beneficio de trabajar con Git. La secuencia del flujo de trabajo que hemos visto es algo así como lo mostrado en la Figura 5-15.

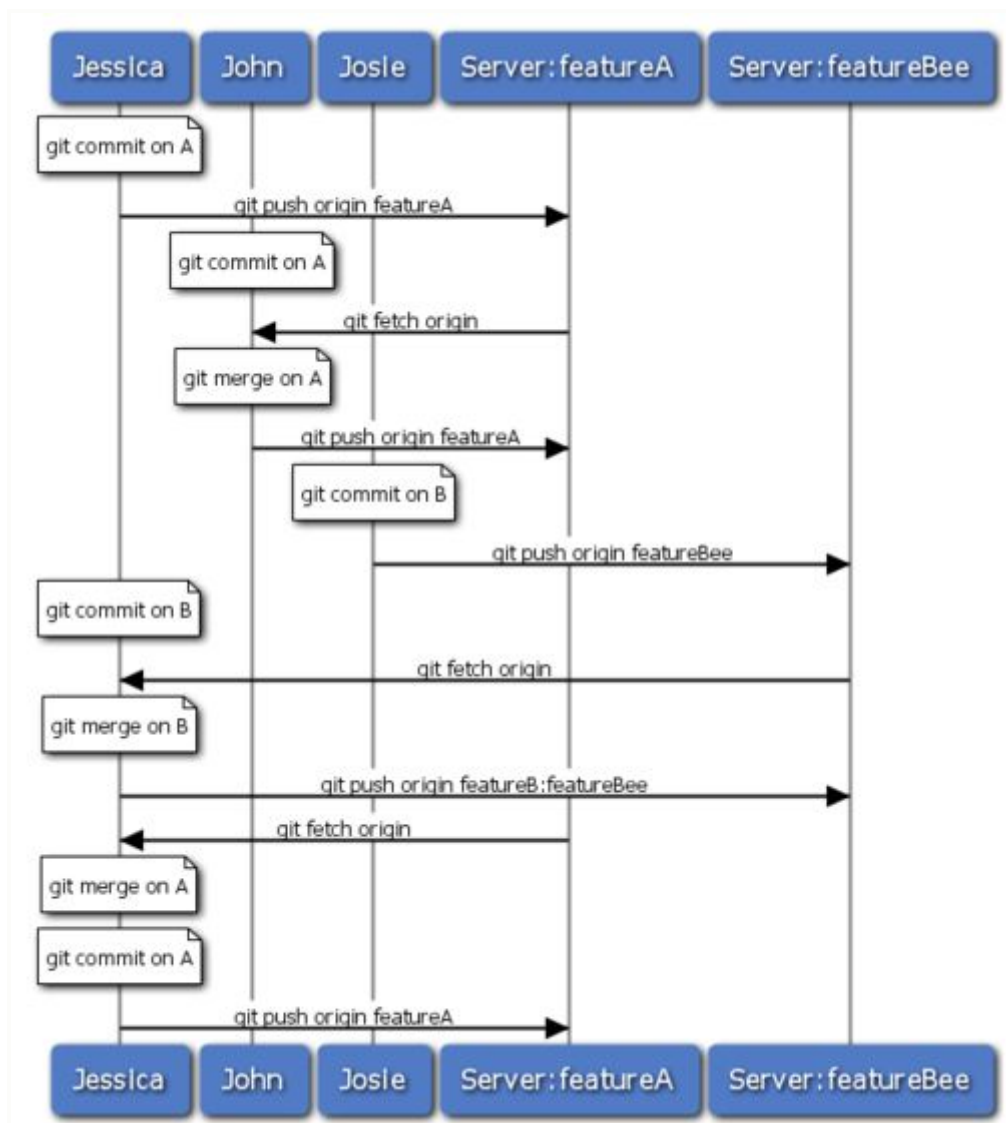


Figura 5-15. Secuencia básica de este flujo de trabajo en equipo gestionado.

### Pequeño Proyecto Público

Contribuir a proyectos públicos es ligeramente diferente. Ya que no tendrás permisos para actualizar ramas directamente sobre el proyecto, has de enviar el trabajo a los gestores de otra manera. En este primer ejemplo, veremos cómo contribuir a través de bifurcaciones (forks) en servidores Git que soporten dichas bifurcaciones. Los sitios como [repo.or.cz](https://repo.or.cz) ó GitHub permiten realizar esto, y muchos gestores de proyectos esperan este estilo de contribución. En la siguiente sección, veremos el caso de los proyectos que prefieren aceptar contribuciones a través del correo electrónico.

Para empezar, probablemente desees clonar el repositorio principal, crear una rama puntual para el parche con el que piensas contribuir, y ponerte a trabajar sobre ella. La secuencia será algo parecido a esto:

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

Puedes querer utilizar 'rebase -i' para reducir tu trabajo a una sola confirmación de cambios (commit), o reorganizar el trabajo en las diversas confirmaciones para facilitar la revisión de parche por parte del gestor del proyecto --ver el Capítulo 6 para más detalles sobre reorganizaciones interactivas--.

Cuando el trabajo en tu rama puntual está terminado y estás listo para enviarlo al gestor del proyecto, vete a la página del proyecto original y clicas sobre el botón "Fork" (bifurcar), para crear así tu propia copia editable del proyecto. Tendrás que añadir la URL a este nuevo repositorio como un segundo remoto, y en este caso lo denominaremos 'myfork':

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

Tu trabajo lo enviarás (push) a este segundo remoto. Es más sencillo enviar (push) directamente la rama puntual sobre la que estás trabajando, en lugar de fusionarla (merge) previamente con tu rama principal y enviar esta última. Y la razón para ello es que, si el trabajo no es aceptado o se integra solo parcialmente, no tendrás que rebobinar tu rama principal. Si el gestor del proyecto fusiona (merge), reorganiza (rebase) o integra solo parcialmente tu trabajo, aún podrás recuperarlo de vuelta a través de su repositorio:

```
$ git push myfork featureA
```

Tras enviar (push) tu trabajo a tu copia bifurcada (fork), has de notificárselo al gestor del proyecto. Normalmente se suele hacer a través de una solicitud de recuperación/integración (pull request). La puedes generar directamente desde el sitio web, --GitHub tiene un botón "pull

request" que notifica automáticamente al gestor--, o puedes lanzar el comando 'git request-pull' y enviar manualmente su salida por correo electrónico al gestor del proyecto.

Este comando 'request-pull' compara la rama base donde deseas que se integre tu rama puntual y el repositorio desde cuya URL deseas que se haga, para darte un resumen de todos los cambios que deseas integrar. Por ejemplo, si Jessica quiere enviar a John una solicitud de recuperación, y ha realizado dos confirmaciones de cambios sobre la rama puntual que desea enviar, lanzará los siguientes comandos:

```
$ 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(-)
```

Esta salida puede ser enviada al gestor del proyecto, --le indica el punto donde se ramificó, resume las confirmaciones de cambio, y le dice desde dónde recuperar estos cambios--.

En un proyecto del que no seas gestor, suele ser más sencillo tener una rama tal como 'master' siguiendo siempre a la rama 'origin/master'; mientras realizas todo tu trabajo en otras ramas puntuales, que podrás descartar fácilmente en caso de que alguna de ellas sea rechazada.

Manteniendo el trabajo de distintos temas aislados en sus respectivas ramas puntuales, te facilitas también el poder reorganizar tu trabajo si la cabeza del repositorio principal se mueve mientras tanto y tus confirmaciones de cambio (commits) ya no se pueden integrar limpiamente. Por ejemplo, si deseas contribuir al proyecto en un segundo tema, no continúes trabajando sobre la rama puntual que acabas de enviar; comienza una nueva rama puntual 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
```

De esta forma, cada uno de los temas está aislado dentro de un silo, --similar a una cola de parches--; permitiendote reescribir, reorganizar y modificar cada uno de ellos sin interferir ni crear interdependencias entre ellos.

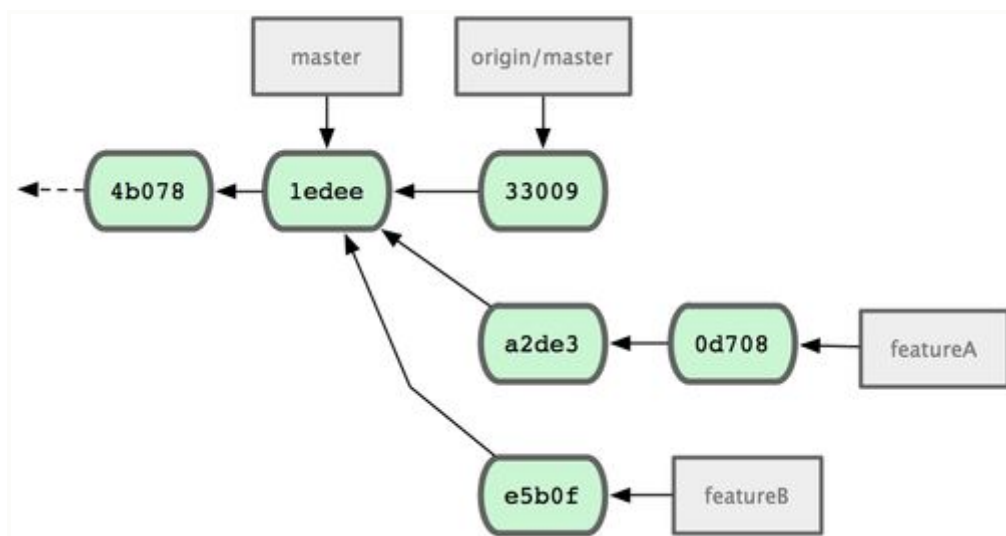


Figura 5-16. Historial inicial con el trabajo de la funcionalidad B.

Supongamos que el gestor del proyecto ha recuperado e integrado un grupo de otros parches y después lo intenta con tu primer parche, viendo que no se integra limpiamente. En este caso, puedes intentar reorganizar (rebase) tu parche sobre 'origin/master', arreglar los conflictos y volver a enviar tus cambios:

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

Esto reescribe tu historial, quedando como se vé en la Figura 5-17.

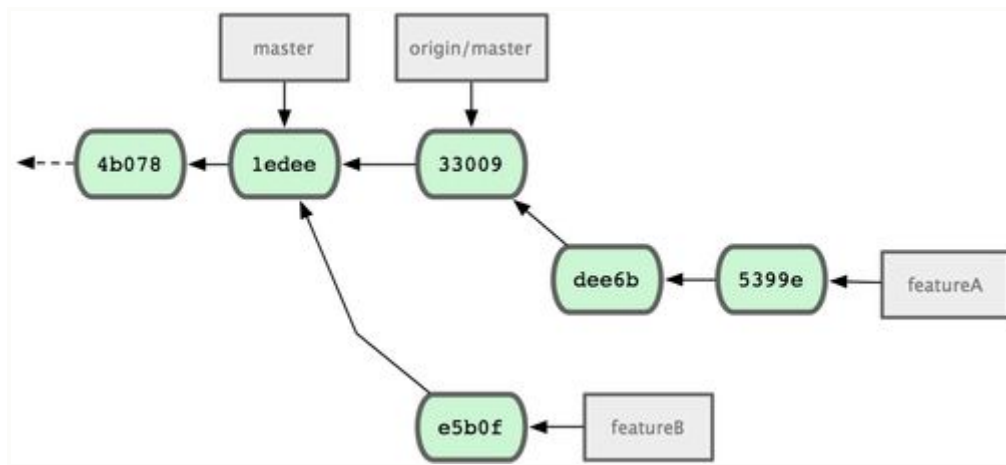


Figura 5-17. Historial tras el trabajo en la funcionalidad A.

Debido a que has reorganizado (rebase) tu rama de trabajo, tienes que indicar la opción '-f' en tu comando de envío (push), para permitir que la rama 'featureA' del servidor sea reemplazada por una confirmación de cambios (commit) que no es hija suya. Una alternativa podría ser el enviar (push) este nuevo trabajo a una rama diferente del servidor (por ejemplo a 'featureAv2').

Vamos a ver otro posible escenario: el gestor del proyecto ha revisado tu trabajo en la segunda rama y le ha gustado el concepto, pero desea que cambies algunos detalles de la implementación. Puedes aprovechar también esta oportunidad para mover el trabajo y actualizarlo sobre la actual rama 'master' del proyecto. Para ello, inicias una nueva rama basada en la actual 'origin/master', aplicas (squash) sobre ella los cambios de 'featureB', resuelves los posibles conflictos que se pudieran presentar, realizas los cambios en los detalles, y la envías de vuelta 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' coge todo el trabajo en la rama fusionada y lo aplica, en una sola confirmación de cambios sin fusión (no-merge commit), sobre la rama en la que estés situado. La opción '--no-commit' indica a Git que no debe registrar automáticamente una confirmación de cambios. Esto te permitirá el aplicar todos los cambios de la otra rama y después hacer más cambios, antes de guardarlos todos ellos en una nueva confirmación (commit).

En estos momentos, puedes notificar al gestor del proyecto que has realizado todos los cambios solicitados y que los puede encontrar en tu rama 'featureBv2' (ver Figura 5-18).

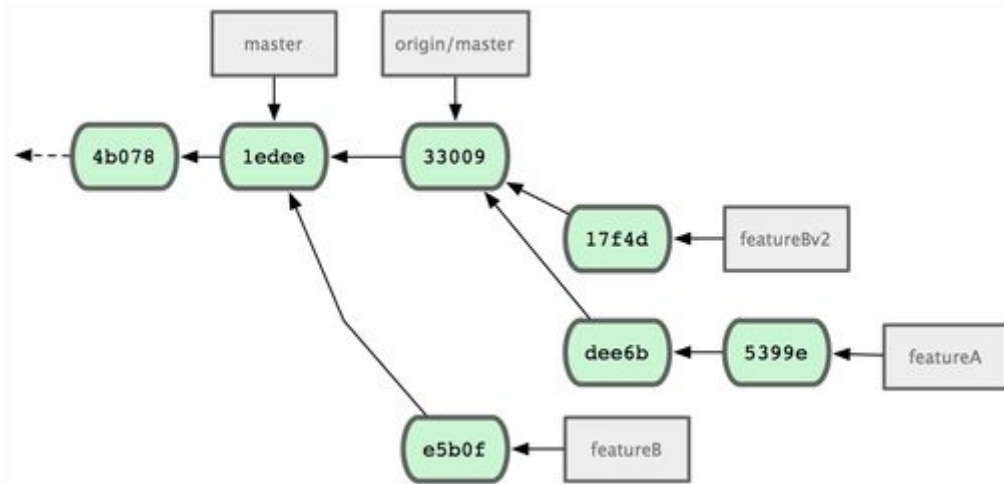


Figura 5-18. Historial tras el trabajo en la versión 2 de la funcionalidad B.

## Proyecto Público Grande

Muchos grandes proyectos suelen tener establecidos los procedimientos para aceptar parches; --es necesario que compruebes las normas específicas para cada proyecto, ya que pueden variar de uno a otro--. De todas formas, muchos de los proyectos públicos más grandes aceptan parches a través de una lista de correo electrónico, por lo que veremos un ejemplo de dicho procedimiento.

El flujo de trabajo es similar a los casos de uso vistos anteriormente; --creando ramas puntuales para cada serie de parches en los que vayas a trabajar--. La diferencia está en la forma de enviarlos al proyecto. En lugar de bifurcar (fork) el proyecto y enviar a tu propia copia editable, generarás correos electrónicos para cada serie de parches y os enviarás a la lista de correo.

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

Tienes dos confirmaciones de cambios (commits) a enviar a la lista de correo. Utilizarás el comando `git format-patch` para generar archivos formateados para poder ser enviados por correo electrónico. Este comando convierte cada confirmación de cambios (commit) en un



mensaje de correo; con la primera línea del mensaje de confirmación puesto como asunto, y el resto del mensaje mas el parche como cuerpo. Lo bonito de este procedimiento es que, al aplicar un parche desde un correo electrónico generado por **format-patch**, se preserva íntegramente la información de la confirmación de cambios (commit). Lo veremos más adelante, en la próxima sección, cuando veamos como aplicarlos:

```
$ 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** lista los nombres de los archivos de parche que crea. La opción **-M** indica a Git que ha de mirar por si hay algo renombrado. Los archivos serán algo como:

```
$ 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 20Limit log functionality to the first 20Limit 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')
--
```

### 1.6.2.rc1.20.g8c5b.dirty

Puedes incluso editar esos archivos de parche, para añadirles más información , específica para la lista de correo, y que no desees mostrar en el propio mensaje de la confirmación de cambios. Si añades texto entre la línea que comienza por `---` y el comienzo del parche (la línea `lib/simplegit.rb`). Los desarrolladores de la lista de correo podrán leerlo. Pero será ignorado al aplicar el parche al proyecto.

Para enviar estos archivos a la lista de correo, puedes tanto pegar directamente el archivo en tu programa de correo electrónico, como enviarlo a través de algún programa basado en línea de comandos. Pegar el texto directamente suele causar problemas de formato. Especialmente con los clientes de correo más "inteligentes", que no preservan adecuadamente los saltos de línea y otros espaciados. Afortunadamente, Git suministra una herramienta que nos puede ser de gran ayuda para enviar parches correctamente formateados a través de protocolo IMAP, facilitándonos así las cosas. Voy a indicar cómo enviar un parche usando Gmail, que da la casualidad de que es el agente de correo utilizado por mí. En el final del anteriormente citado documento, [Documentation/SubmittingPatches](#), puedes leer instrucciones detalladas para otros agentes de correo.

Lo primero es configurar correctamente el apartado imap de tu archivo `~/.gitconfig`. Puedes ir ajustando cada valor individualmente, a través de comandos `git config`; o puedes añadirlos todos manualmente. Pero, al final, tu archivo de configuración ha de quedar más o menos como esto:

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

Las dos últimas líneas probablemente no sean necesarias si tu servidor IMAP no utiliza SSL; y, en ese caso, el valor para `host` deberá de ser `imap://` en lugar de `imaps://`. Cuando tengas esto ajustado, podrás utilizar el comando `git send-email` para poner series de parches en la carpeta de borradores (Drafts) de tu servidor IMAP:

```
$ 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

Tras esto, Git escupirá una serie de información de registro, con pinta más o menos como esta:

```
(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 estos momentos, deberías poder ir a tu carpeta de borradores (Drafts), cambiar el destinatario (To) para apuntar a la lista de correo donde estés enviando el parche, puede que poner en copia (CC) al gestor o persona responsable, y enviar el mensaje.

## **Recapitulación**

En esta sección hemos visto unos cuantos de los flujos de trabajo más habituales para lidiar con distintos tipos de proyectos Git. Y hemos introducido un par de nuevas herramientas para ayudarte a gestionar estos procesos. A continuación veremos cómo trabajar en el otro lado de la moneda: manteniendo y gestionando un proyecto Git. Vas a aprender cómo ser un dictador benevolente o un gestor de integración.

## 5.3 Git en entornos distribuidos - Gestionando un proyecto

### Gestionando un proyecto

Además de conocer cómo contribuir de forma efectiva a un proyecto, es posible que desees saber también cómo mantener uno. Lo cual implicará saber aceptar y aplicar parches generados vía `format-patch`, enviados a tí a través de correo electrónico; o saber integrar cambios realizados en ramas de repositorios que añadirás como remotos a tu proyecto. Tanto si gestionas un repositorio canónico, como si deseas colaborar verificando o aprobando parches, necesitas saber cómo aceptar trabajo de otros de la forma más clara para tus contribuyentes y más sostenible para tí a largo plazo.

### Trabajando con Ramas Puntuales

Cuando estás pensando en integrar nuevo trabajo, suele ser buena idea utilizar una rama puntual para cada tema concreto --una rama temporal creada específicamente para trabajar dicho tema--. De esta forma, es sencillo tratar cada parche de forma individualizada y poder "aparcar" uno concreto cuando no trabajamos en él, hasta cuando volvamos a tener tiempo para retomarlo. Si creas los nombres de ramas basandolos en el tema sobre el que vas a trabajar, por ejemplo 'ruby client' o algo así de descriptivo, podrás recordar de qué iba cada rama en caso de que la abandones por un tiempo y la retomes más tarde. La persona gestora del proyecto Git suele tender a nombrar cada rama de forma parecida --por ejemplo 'sc/ruby client', donde sc es la abreviatura para la persona que ha contribuido con ese trabajo--. Como recordarás, la forma de crear una rama basandola en tu rama master es:

```
$ git branch sc/ruby_client master
```

O, si deseas crearla y saltar inmediatamente a ella, puedes también utilizar la opción '-b' del comando 'checkout':

```
$ git checkout -b sc/ruby_client master
```

Tras esto, estarás listo para añadir tu trabajo a esa rama puntual y ver si deseas o no fusionarla luego con alguna otra de tus ramas de más largo recorrido.

## Aplicar parches recibidos por correo electronico

Si vas a integrar en tu proyecto un parche recibido a través de un correo electrónico. Antes de poder evaluarlo, tendrás que incorporarlo a una de tus ramas puntuales. Tienes dos caminos para incorporar un parche recibido por correo electronico: usando el comando 'git apply' o usando el comando 'git am'.

### **Incorporando un parche con apply**

Si recibes un parche de alguien que lo ha generado con el comando 'git diff' o con un comando 'diff' de Unix, puedes incorporarlo con el comando 'git apply'. Suponiendo que has guardado el parche en '/tmp/patch-ruby-client.patch', puedes incorporarlo con una orden tal como:

```
$ git apply /tmp/patch-ruby-client.patch
```

Esto modificará los archivos en tu carpeta de trabajo. Es prácticamente idéntico a lanzar el comando 'patch -p1', aunque es más paranoico y acepta menos coincidencias aproximadas. Además, es capaz de manejar adicciones, borrados o renombrados de archivos, si vienen en formato 'git diff'. Mientras que 'patch' no puede hacerlo. Por ultimo, citar que 'git apply' sigue un modelo de "aplicar todo o abortar todo", incorporando todos los cambios o no incorporando ninguno. Mientras que 'patch' puede incorporar cambios parcialmente, dejando tu carpeta de trabajo en un estado inconsistente. 'git apply' es, de lejos, mucho más paranoico que 'patch'. Nunca creará una confirmación de cambios (commit) por tí, --tras ejecutar el comando, tendrás que preparar (stage) y confirmar (commit) manualmente todos los cambios introducidos--.

Tambien puedes utilizar 'git apply' para comprobar si un parche se puede incorporar limpiamente; antes de intentar incorporarlo. Puedes lanzar el comando 'git apply --check':

```
$ 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 obtienes una salida vacia, el parche se podrá incorporar limpiamente. Además, este comando retorna con un status no-cero en caso de fallar la comprobación, por lo que puedes utilizarlo en scripts si lo deseas.

### **Incorporando un parche con am**

Si la persona que contribuye es usuaria de Git y conoce lo suficiente como para utilizar el comando 'format-patch' al generar su parche, tendrás mucho camino recorrido al incorporarlo; ya que el parche traerá consigo información sobre el o la autora, además de un mensaje de

confirmación de cambios. Si puedes, anima a tus colaboradoras a utilizar 'format-patch' en lugar de 'diff' cuando vayan a generar parches. Solo deberías utilizar 'git apply' en caso de parches antiguos y similares.

Para incorporar un parche generado con 'format-patch', utilizarás el comando 'git am'.

Técnicamente, 'git am' se construyó para leer un archivo de buzón de correo (mbox file), que no es más que un simple formato de texto plano para almacenar uno o varios mensajes de correo electrónico en un solo 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 20Limit log functionality to the first 20Limit log
functionality to the first 20
```

Esto es el comienzo de la salida del comando format-patch visto en la sección anterior. Es también un formato válido para un mbox. Si alguien te ha enviado correctamente un parche utilizando 'git send-email', y te lo has descargado a un formato mbox; podrás indicar dicho archivo mbox al comando 'git am', y este comenzará a incorporar todos los parches que encuentre dentro. Si tienes un cliente de correo electrónico capaz de guardar varios mensajes en formato mbox, podrás guardar series completas de parches en un mismo archivo; y luego usar 'git am' para irlos incorporando secuencialmente.

Sin embargo, si alguien sube su archivo de parche a un sistema de gestión de peticiones de servicio o similar; tendrás que descargarlo a un archivo local en tu disco y luego indicar ese archivo local al comando 'git am':

```
$ git am 0001-limit-log-function.patch
Applying: add limit to log function
```

Observarás que, tras incorporarlo limpiamente, crea automáticamente una nueva confirmación de cambios (commit). La información sobre el autor o autora la recoge de las cabeceras 'From' (Remitente) y 'Date' (Fecha). Y el mensaje para la confirmación (commit) lo recoge de 'Subject' (Asunto) y del cuerpo del correo electrónico. Por ejemplo, si consideramos el parche incorporado desde el mbox del ejemplo que acabamos de mostrar; la confirmación de cambios (commit) generada será algo como:

```
$ 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  
Limit log functionality to the first 20  
Limit log functionality to the first 20

El campo 'Commit' muestra la persona que ha incorporado el parche y cuándo lo ha incorporado. El campo 'Author' muestra la persona que ha creado originalmente el parche y cuándo fue creado este.

Pero también podría suceder que el parche no se pudiera incorporar limpiamente. Es posible que tu rama principal diverja demasiado respecto de la rama sobre la que se construyó el parche; o que el parche tenga dependencias respecto de algún otro parche anterior que aún no hayas incorporado. En ese caso, el proceso 'git am' fallará y te preguntará qué deseas 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 pondrá marcadores de conflicto en cualquier archivo con problemas, de forma similar a como lo haría una operación de fusión (merge) o de reorganización (rebase). Y resolverás los problemas de la misma manera: editar el archivo para resolver los conflictos, prepararlo (stage), y lanzar '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 deseas más inteligencia por parte de Git al resolver conflictos, puedes pasarle la opción '-3', para que intente una fusión a tres bandas (three-way merge). Esta opción no se usa por defecto, porque no funcionará en caso de que la confirmación de cambios en que el parche dice estar basado no esté presente en tu repositorio. Sin embargo, si tienes dicha confirmación de cambios (commit), --si el parche está basado en una confirmación pública--, entonces la opción '-3' suele ser mucho más avispada cuando incorporas 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, estamos intentando incorporar un parche que ya tenemos incorporado. Sin la opción '-3', tendríamos problemas.

Al aplicar varios parches desde un mbox, puedes lanzar el comando 'am' en modo interactivo; haciendo que se detenga en cada parche y preguntandote si aplicarlo o no:

```
$ 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
```

Es una utilidad interesante si tienes almacenados unos cuantos parches, porque puedes ir revisando previamente cada parche y aplicarlos selectivamente.

Cuando tengas integrados y confirmados todos los parches relativos al tema puntual en que estas trabajando, puedes plantearte cómo y cuándo lo vas a integrar en alguna otra rama de más largo recorrido.

## Recuperando ramas remotas



Si recibes una contribución de un usuario que ha preparado su propio repositorio, ha guardado unos cuantos cambios en este, y luego te ha enviado la URL del repositorio y el nombre de la rama remota donde se encuentran los cambios. Puedes añadir dicho repositorio como un remoto y fusionar los cambios localmente.

Por ejemplo, si Jessica te envía un correo electrónico comentándote que tiene una nueva e interesante funcionalidad en la rama 'ruby-client' de su repositorio. Puedes probarla añadiendo el remoto correspondiente 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 vuelva a enviarte otro correo electrónico avisándote de otra gran funcionalidad que ha incorporado, puedes recuperarla (fetch y checkout) directamente, porque tienes el remoto ya definido.

Es muy útil cuando trabajas regularmente con una persona. En cambio, si alguien tiene un solo parche para enviarte, una sola vez, puede ser más efectivo aceptarlo directamente por correo electrónico; en lugar de pedir a todo el mundo que tenga cada uno su propio servidor y tener nosotros que estar continuamente añadiendo y quitando remotos para cada parche. También es muy posible que no quieras tener cientos de remotos, cada uno contribuyendo tan solo con un parche o dos. De todas formas, los scripts y los servicios albergados pueden hacerte la vida más fácil en esto, --todo depende de cómo desarrolles tú y de cómo desarrollan las personas que colaboran contigo--.

Otra ventaja de esta forma de trabajar es que recibes también el histórico de confirmaciones de cambio (commits). A pesar de poder seguir teniendo los habituales problemas con la fusión, por lo menos conoces en qué punto de tu historial han basado su trabajo. Por defecto, se aplicará una genuina fusión a tres bandas, en lugar de tener que poner un '-3' y esperar que el parche haya sido generado a partir de una confirmación de cambios (commit) pública a la que tengas tú también acceso.

Si no trabajas habitualmente con una persona, pero deseas recuperar de ella por esta vía, puedes indicar directamente el URL del repositorio remoto en el comando 'git pull'. Esto efectúa una recuperación (pull) puntual y no conserva la URL como una referencia remota:

```
$ git pull git://github.com/onetimeguy/project.git
From git://github.com/onetimeguy/project
```

\* branch            HEAD        -> FETCH\_HEAD  
Merge made by recursive.

## Revisando lo introducido

Ahora que tienes una rama puntual con trabajo aportado por otras personas. Tienes que decidir lo que deseas hacer con él. En esta sección revisaremos un par de comandos, que te ayudarán a ver exactamente los cambios que introducirás si fusionas dicha rama puntual con tu rama principal.

Suele ser útil revisar todas las confirmaciones de cambios (commits) que esten en esta rama, pero no en tu rama principal. Puedes excluir de la lista las confirmaciones de tu rama principal añadiendo la opción '--not' delante del nombre de la rama. Por ejemplo, si la persona colaboradora te envía dos parches y tu creas una rama 'contrib' donde aplicar dichos parches; puedes lanzar algo como 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 en detalle los cambios introducidos por cada confirmación (commit), recuerda que pasando la opción '-p' al comando 'git log', obtendrás un listado extendido con las diferencias introducidas por cada confirmación.

Para ver plenamente todas las diferencias y lo que sucederá realmente si fusionas esta rama puntual con otra rama, tendrás que utilizar un pequeño truco para obtener los resultados correctos. Puedes pensar en lanzar esto:

```
$ git diff master
```

Este comando te dará las diferencias, pero puede ser engañoso. Si tu rama 'master' ha avanzado con respecto al momento en que se creó la rama puntual a partir de ella, puedes obtener resultados realmente extraños. Debido a que Git compara directamente las instantáneas de la última confirmación de cambios en la rama puntual donde te encuentras y en la rama 'master'. Por ejemplo, si en la rama 'master' habías añadido una línea a un archivo, la comparación directa de instantáneas te llevará a pensar que la rama puntual va a borrar dicha línea.

Si 'master' es un ancestro directo de tu rama puntual, no tendrás problemas. Pero si los historiales de las dos ramas son divergentes, la comparación directa de diferencias dará la apariencia de estar añadiendo todo el material nuevo en la rama puntual y de estar borrando todo el material nuevo en la rama 'master'.

Y lo que realmente querías ver eran los cambios introducidos por la rama puntual, --el trabajo que vas a introducir si la fusionas con la rama 'master'--. Lo puedes hacer indicando a Git que compare la última confirmación de cambios en la rama puntual, con el más reciente ancestro común que tenga esta respecto de la rama 'master'.

Técnicamente, lo puedes hacer descubriendo tu mismo dicho ancestro común y lanzando la comprobación de diferencias respecto de él:

```
$ git merge-base contrib master  
36c7dba2c95e6bbb78dfa822519ecfec6e1ca649  
$ git diff 36c7db
```

Pero esto no es lo más conveniente. De ahí que Git suministre otro atajo para hacerlo: la sintaxis del triple-punto. Estando en el contexto del comando 'diff', puedes indicar tres puntos entre los nombres de las dos ramas; para comparar entre la última confirmación de cambios de la rama donde estás y la respectiva confirmación común con la otra rama:

```
$ git diff master...contrib
```

Este comando mostrará únicamente el trabajo en tu actual rama puntual que haya sido introducido a partir de su ancestro común con la rama 'master'. Es una sintaxis muy útil, que merece recordar.

## Integrando el trabajo aportado por otros

Cuando todo el trabajo presente en tu rama puntual esté listo para ser integrado en una rama de mayor rango, la cuestión es cómo hacerlo. Yendo aún más lejos, ¿cual es el sistema de trabajo que

deseas utilizar para el mantenimiento de tu proyecto? Tienes bastantes opciones, y vamos a ver algunas de ellas.

### Fusionando flujos de trabajo

Una forma simple de trabajar es fusionandolo todo en tu rama 'master'. En este escenario, tienes una rama 'master' que contiene, principalmente, código estable. Cuando en una rama puntual tienes trabajo ya terminado o contribuciones ya verificadas de terceros, los fusionas en tu rama 'master', borras la rama puntual, y continuas trabajando en otra/s rama/s. Si, tal y como se muestra en la Figura 5-19, tenemos un repositorio con trabajos en dos ramas, denominadas 'ruby client' y 'php client'; y fusionamos primero la rama 'ruby client' y luego la 'php client', obtendremos un historial similar al de la Figura 5-20.

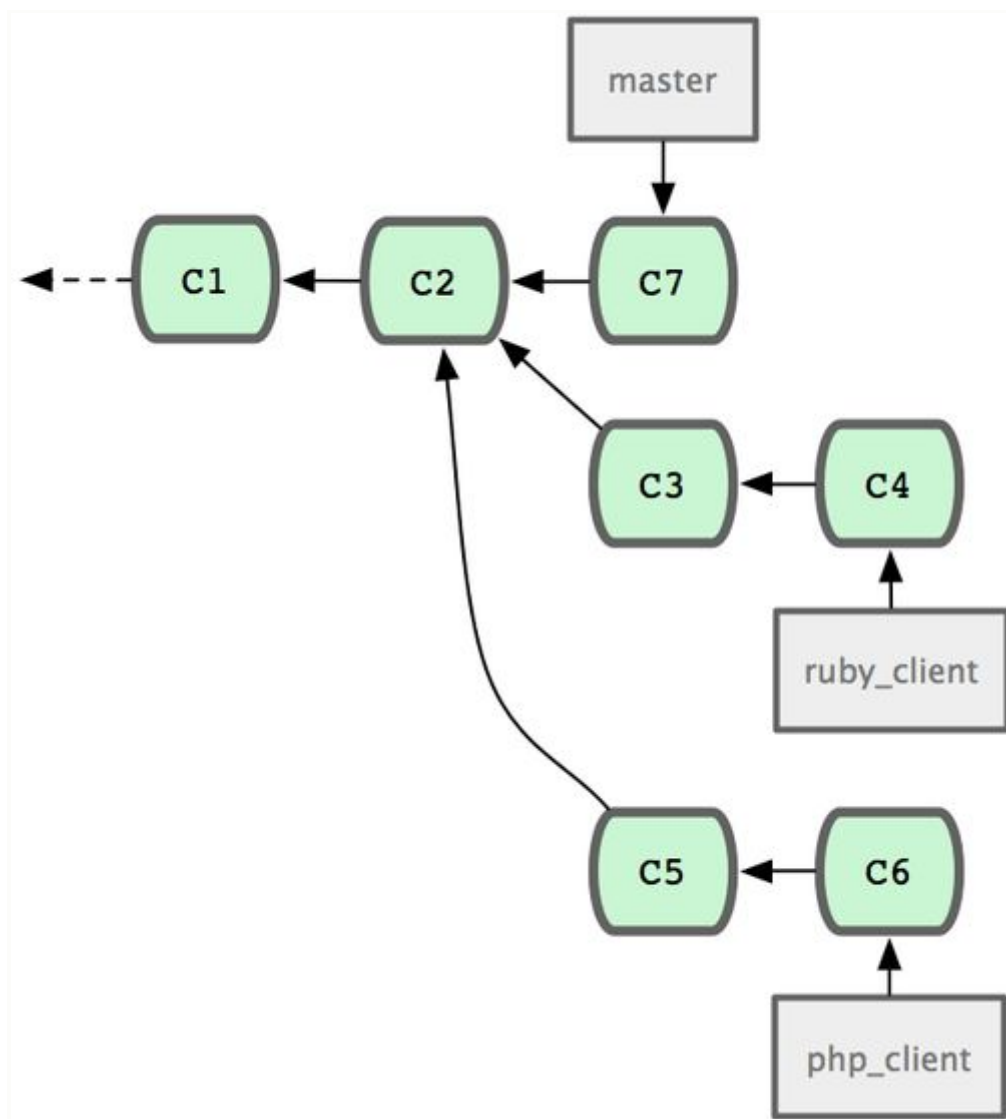


Figura 5-19. Historial con varias ramas puntuales.

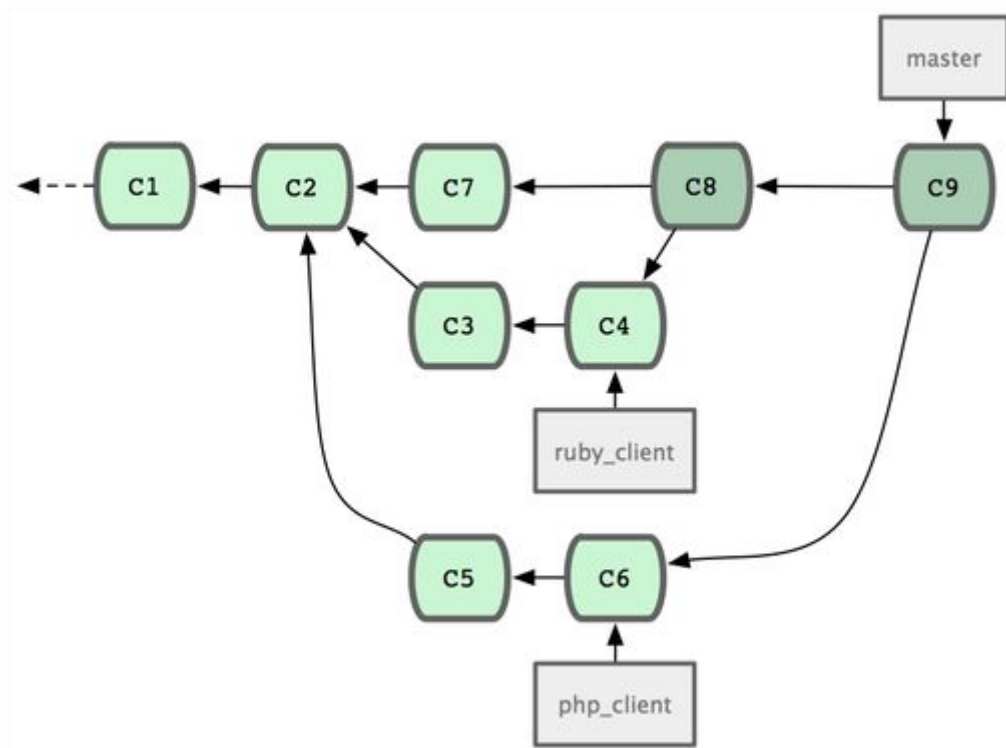


Figura 5-20. Tras fusionar una rama puntual.

Este es probablemente el flujo de trabajo más sencillo. Pero puede dar problemas cuando estás tratando con grandes repositorios o grandes proyectos.

Teniendo muchos desarrolladores o proyectos muy grandes, muy posiblemente desees utilizar un ciclo con por lo menos dos fases. En este escenario, se dispone de dos ramas de largo recorrido: 'master' y 'develop'. La primera de ellas, 'master', será actualizada únicamente por los lanzamientos de código muy estable. La segunda rama, 'develop', es donde iremos integrando todo el código nuevo. Ambas ramas se enviarán periódicamente al repositorio público. Cada vez que tengas una nueva rama puntual lista para integrar (Figura 5-21), la fusionarás en la rama 'develop'. Y cuando marques el lanzamiento de una versión estable, avanzarás la rama 'master' hasta el punto donde la rama 'develop' se encuentre en ese momento (Figura 5-23).

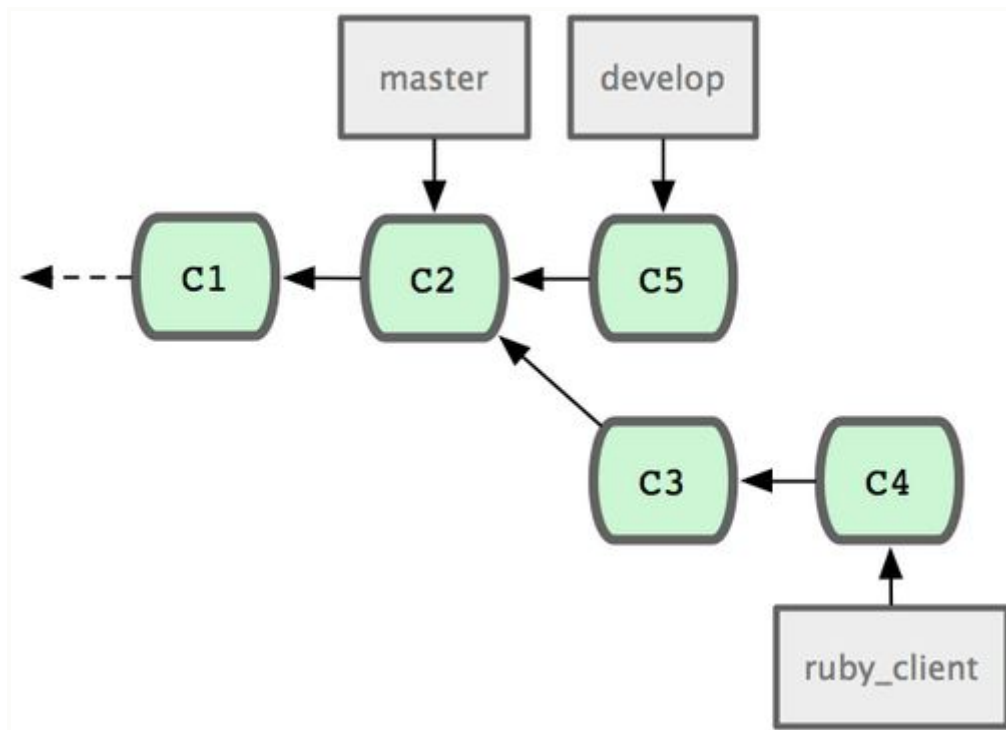


Figura 5-21. Antes de fusionar una rama puntual.

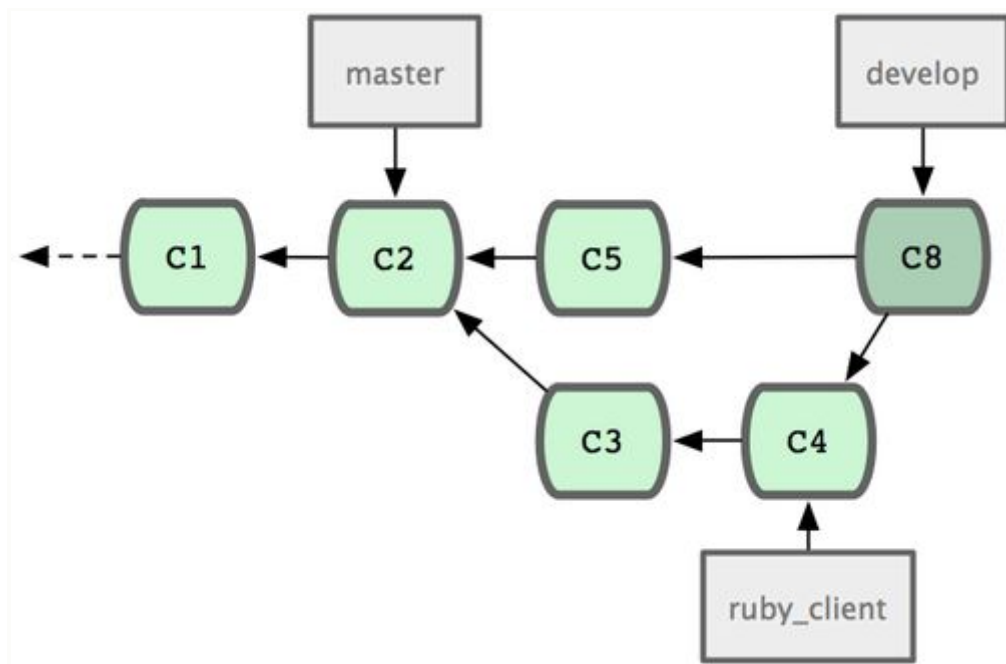


Figura 5-22. Tras fusionar una rama puntual.

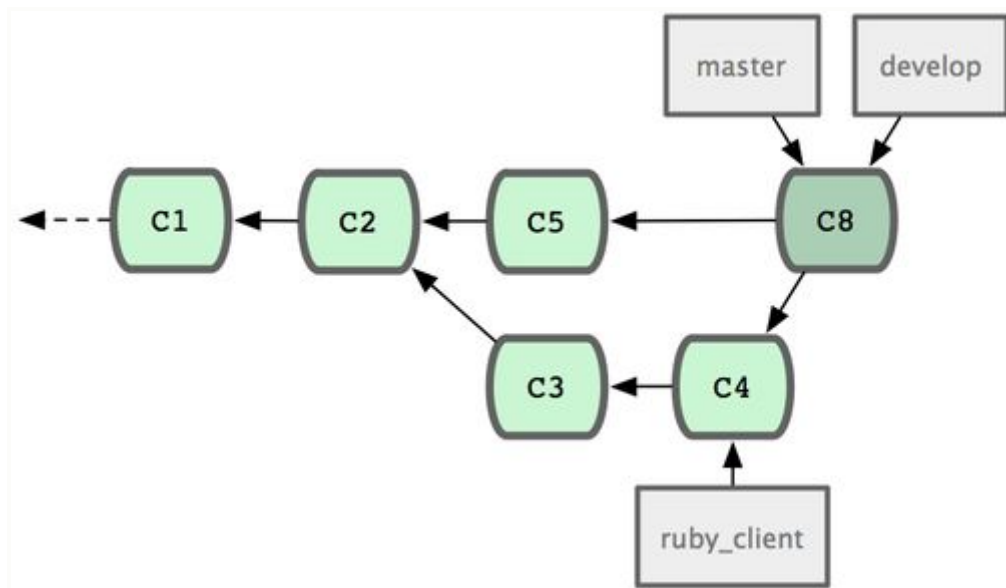


Figura 5-23. Tras un lanzamiento puntual.

De esta forma, cuando alguien clone el repositorio de tu proyecto, podrá recuperar (checkout) y mantener actualizadas tanto la última versión estable como la versión con el material más avanzado; en las ramas 'master' y 'develop', respectivamente. Puedes continuar ampliando este concepto, disponiendo de una rama 'integrate' donde ir fusionando todo el trabajo entre sí. A continuación, cuando el código en dicha rama sea estable y pase todas las pruebas, la fusionarás con la rama 'develop'; y, cuando se demuestre que permanece estable durante un cierto tiempo, avanzarás la rama 'master' hasta ahí.

### Flujos de trabajo con grandes fusiones

El proyecto Git tiene cuatro ramas de largo recorrido: 'master', 'next', 'pu' (proposed updates) para el trabajo nuevo, y 'maint' (maintenance) para trabajos de mantenimiento de versiones previas. A medida que vamos introduciendo nuevos trabajos de las personas colaboradoras, estos se van recolectando en ramas puntuales en el repositorio de una persona gestora; de forma similar a como se ha ido describiendo (ver Figura 5-24). En un momento dado, las funcionalidades introducidas se evalúan; comprobando si son seguras y si están preparadas para los consumidores; o si, por el contrario, necesitan dedicarles más trabajo. Las funcionalidades que resultan ser seguras y estar preparadas se fusionan (merge) en la rama 'next'; y esta es enviada (push) al repositorio público, para que cualquiera pueda probarlas.

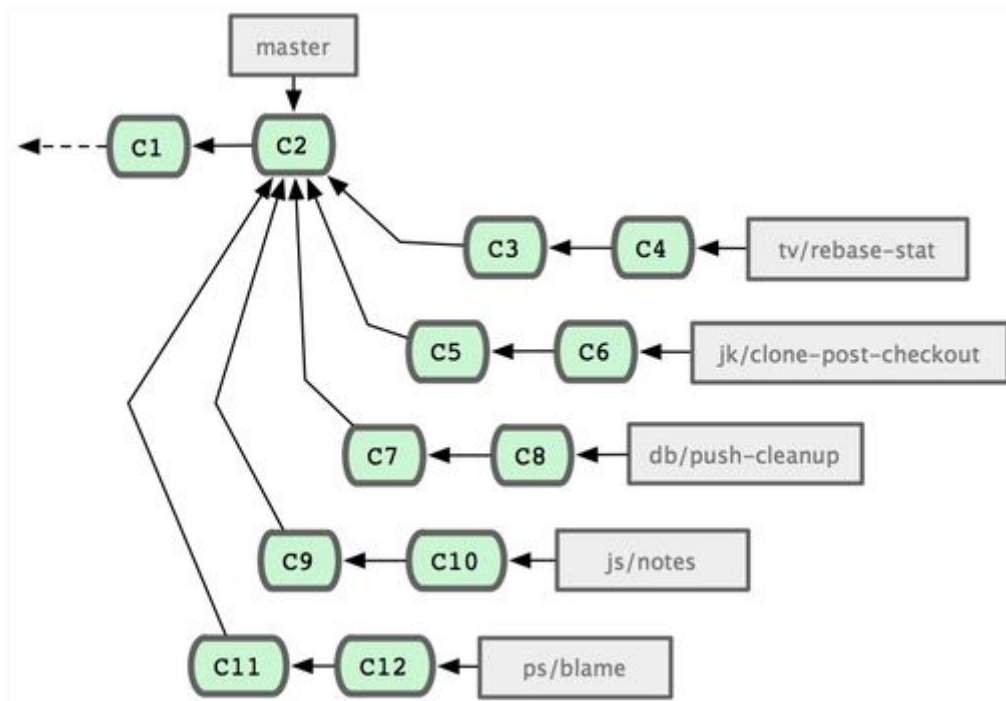


Figura 5-24. Gestionando complejas series de ramas puntuales paralelas con funcionalidades varias.

Si las funcionalidades necesitan ser más trabajadas, se fusionan (merge) en la rama 'pu'. Y cuando las funcionalidades permanecen totalmente estables, se refusionan en la rama 'master'; componiéndolas desde las funcionalidades en la rama 'next' aún sin promocionar a 'master'. Esto significa que 'master' prácticamente siempre avanza; 'next' se reorganiza (rebase) de vez en cuando; y 'pu' es reorganizada con más frecuencia (ver Figura 5-25).

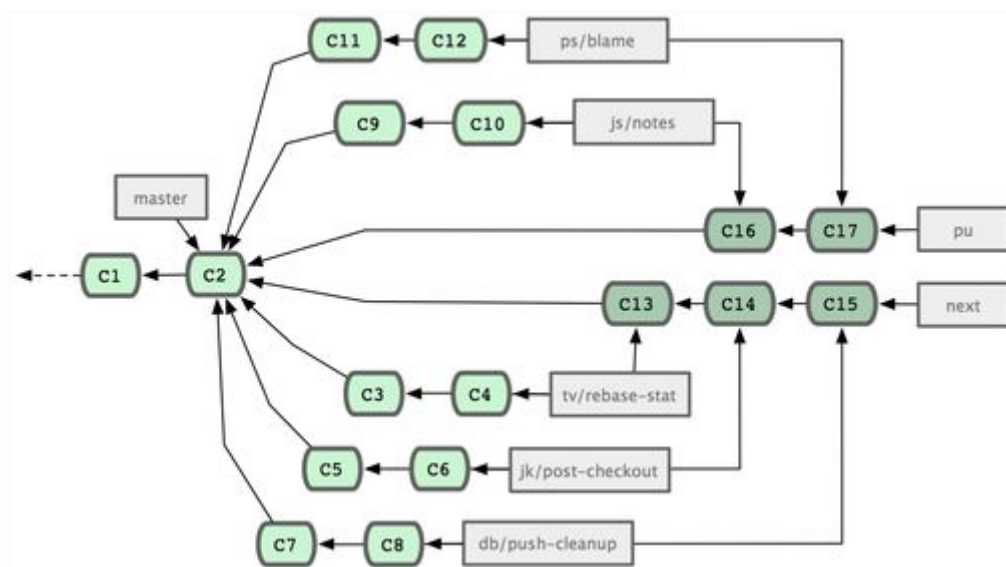




Figura 5-25. Fusionando aportaciones de ramas puntuales en ramas de más largo recorrido.

Una rama puntual se borra del repositorio cuando, finalmente, es fusionada en la rama 'master'. El proyecto Git dispone también de una rama 'maint' que se bifurca (fork) a partir de la última versión ya lanzada; para trabajar en parches, en caso de necesitarse alguna versión intermedia de mantenimiento. Así, cuando clonas el repositorio de Git, obtienes cuatro ramas que puedes recuperar (checkout); pudiendo evaluar el proyecto en distintos estadios de desarrollo, dependiendo de cuán avanzado desees estar o cómo desees contribuir. Y así, los gestores de mantenimiento disponen de un flujo de trabajo estructurado, para ayudarles en el procesado e incorporación de nuevas contribuciones.

### **Flujos de trabajo reorganizando o entresacando**

Otros gestores de mantenimiento, al procesar el trabajo recibido de las personas colaboradoras, en lugar de fusiones (merge), suelen preferir reorganizar (rebase) o entresacar (cherry-pick) sobre su propia rama principal; obteniendo así un historial prácticamente lineal. Cuando desees integrar el trabajo que tienes en una rama puntual, te puedes situar sobre ella y lanzar el comando 'rebase'; de esta forma recompondrás los cambios encima de tu actual rama 'master' (o 'develop' o lo que corresponda). Si funciona, se realizará un avance rápido (fast-forward) en tu rama 'master', y acabarás teniendo un historial lineal en tu proyecto.

El otro camino para introducir trabajo de una rama en otra, es entresacarlo. Entresacar (cherry-pick) en Git es como reorganizar (rebase) una sola confirmación de cambios (commit). Se trata de coger el parche introducido por una determinada confirmación de cambios e intentar reaplicarlo sobre la rama donde te encuentres en ese momento. Puede ser útil si tienes varias confirmaciones de cambios en una rama puntual, y tan solo desees integrar una de ellas; o si tienes una única confirmación de cambios en una rama puntual, y prefieres entresacarla en lugar de reorganizar. Por ejemplo, suponiendo que tienes un proyecto parecido al ilustrado en la Figura 5-26.

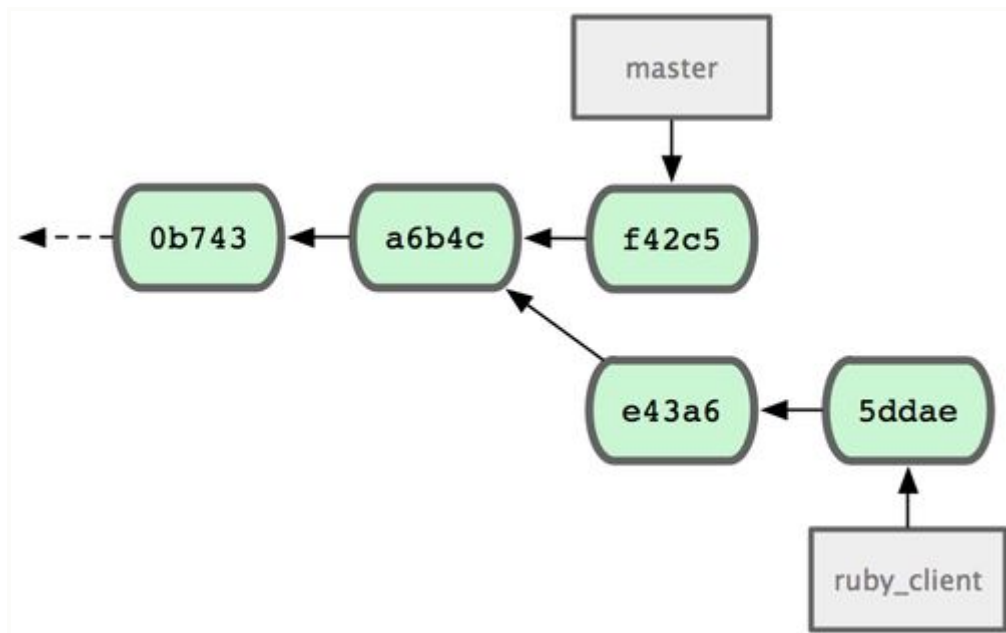


Figura 5-26. Historial de ejemplo, antes de entresacar.

Si deseas integrar únicamente la confirmación 'e43a6' en tu rama 'master', puedes lanzar:

```
$ 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 exactamente el mismo cambio introducido por 'e43a6', pero con un nuevo valor SHA-1 de confirmación; ya que es diferente la fecha en que ha sido aplicado. Tu historial quedará tal como ilustra la Figura 5-27.

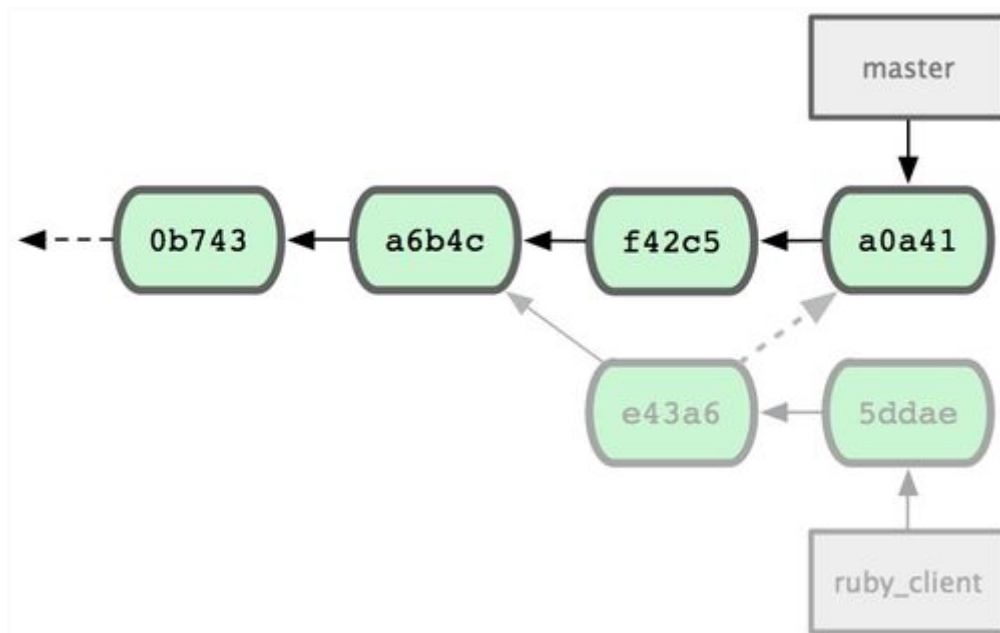


Figura 5-27. Historial tras entresacar una confirmación de cambios de una rama puntual.

Ahora, ya puedes borrar la rama puntual y descartar las confirmaciones de cambios que no deseas integrar.

### Marcando tus lanzamientos de versiones

Cuando decides dar por preparada una versión, probablemente querrás etiquetar dicho punto de algún modo; de tal forma que, más adelante, puedas volver a generar esa versión en cualquier momento. Puedes crear una nueva etiqueta tal y como se ha comentado en el capítulo 2. Si decides firmar la etiqueta como gestor de mantenimientos que eres, el proceso será algo como:

```
$ 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, puedes tener un problema a la hora de distribuir la clave PGP pública utilizada en la firma. Los gestores del proyecto Git han resuelto este problema incluyendo sus claves públicas como un objeto en el repositorio, añadiendo luego una etiqueta apuntando directamente a dicho contenido. Para ello, has de seleccionar cada clave que deseas incluir, 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]
```

Tras esto, puedes importar directamente la clave en la base de datos Git, exportandola y redirigiendola a través del comando 'git hash-object'. Para, de esta forma, escribir un nuevo objeto dentro de Git y obtener de vuelta la firma SHA-1 de dicho objeto.

```
$ gpg -a --export F721C45A | git hash-object -w --stdin  
659ef797d181633c87ec71ac3f9ba29fe5775b92
```

Una vez tengas el contenido de tu clave guardado en Git, puedes crear una etiqueta que apunte directamente al mismo; indicando para ello el nuevo valor SHA-1 que te ha devuelto el objeto 'hash-object':

```
$ git tag -a maintainer-pgp-pub 659ef797d181633c87ec71ac3f9ba29fe5775b92
```

Si lanzas el comando 'git push --tags', la etiqueta 'maintainer-pgp-pub' será compartida por todos. Cualquiera que desee verificar la autenticidad de una etiqueta, no tiene más que importar tu clave PGP, sacando el objeto directamente de la base de datos e importandolo en GPG;

```
$ git show maintainer-pgp-pub | gpg --import
```

De esta forma, pueden utilizar esa clave para verificar todas las etiquetas que firmes. Además, si incluyes instrucciones en el mensaje de etiquetado, con el comando `git show <tag>`, los usuarios podrán tener directrices específicas acerca de la verificación de etiquetas.

## Generando un número de ensamblado

Debido a que Git no dispone de una serie monótona ascendente de números para cada confirmación de cambios (commit), si deseas tener un nombre humanamente comprensible por cada confirmación, has de utilizar el comando 'git describe'. Git te dará el nombre de la etiqueta más cercana, mas el número de confirmaciones de cambios entre dicha etiqueta y la confirmación que estas describiendo, más una parte de la firma SHA-1 de la confirmación:

```
$ git describe master  
v1.6.2-rc1-20-g8c5b85c
```

De esta forma, puedes exportar una instantánea u obtener un nombre comprensible por cualquier persona. Es más, si compilas Git desde código fuente clonado desde el repositorio Git, el comando 'git --version' te dará algo parecido. Si solicitas descripción de una confirmación de cambios (commit) etiquetada directamente con su propia etiqueta particular, obtendrás dicha etiqueta como descripción.

El comando 'git describe' da preferencia a las etiquetas anotativas (etiquetas creadas con las opciones '-a' o '-s'). De esta forma las etiquetas para las versiones pueden ser creadas usando 'git describe', asegurandose el que las confirmaciones de cambios (commit) son adecuadamente nombradas cuando se describen. También puedes utilizar esta descripción para indicar lo que deseas activar (checkout) o mostrar (show); pero realmente estarás usando solamente la parte final de la firma SHA-1 abreviada, por lo que no siempre será válida. Por ejemplo, el kernel de Linux ha saltado recientemente de 8 a 10 caracteres, para asegurar la unicidad de los objetos SHA-1; dando como resultado que los nombres antiguos de 'git describe' han dejado de ser válidos.

### Preparando un lanzamiento de versión

Si quieres lanzar una nueva versión. Una de las cosas que desearas crear es un archivo con la más reciente imagen de tu código, para aquellas pobres almas que no utilizan 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
```

Quien abra ese archivo tarball, obtendrá la más reciente imagen de tu proyecto; puesta bajo una carpeta de proyecto. También puedes crear un archivo zip de la misma manera, tan solo indicando la opción '--format=zip' al comando 'git archive':

```
$ git archive master --prefix='project/' --format=zip > `git describe master`.zip
```

Así, tendrás sendos archivos tarball y zip con tu nueva versión, listos para subirlos a tu sitio web o para ser enviados por correo electrónico a tus usuarios.

## El registro rápido

Ya va siendo hora de enviar un mensaje a tu lista de correo, informando a las personas que desean conocer la marcha de tu proyecto. Una manera elegante de generar rápidamente una lista con los principales cambios añadidos a tu proyecto desde la anterior versión, es utilizando el comando 'git shortlog'. Este comando resume todas las confirmaciones de cambios (commits) en el rango que le indiques. Por ejemplo, si tu último lanzamiento de versión lo fué de 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

Obtendrás un claro resumen de todas las confirmaciones de cambios (commit) desde la versión v1.0.1, agrupadas por autor, y listas para ser incorporadas en un mensaje a tu lista de correo.

# Chapter 6

## Las herramientas de Git

A estas alturas, hemos aprendido la mayoría de los comandos y flujos de trabajo empleados habitualmente a la hora de utilizar, gestionar y mantener un repositorio Git para el control de versiones de código fuente. Se han visto las tareas básicas de seguimiento y confirmación de cambios en archivos. Aprovechando las capacidades del área de preparación (staging area), de las ramas (branches) y de los mecanismos de fusión (merging).

En este capítulo se van a explorar unas cuantas tareas avanzadas de Git. Tareas que, aunque no se utilizan en el trabajo del día a día, en algún momento pueden ser necesarias.

## 6.1 Las herramientas de Git - Selección de confirmaciones de cambios concretas

### **Selección de confirmaciones de cambios concretas**

Git tiene varios modos de seleccionar confirmaciones de cambio o grupos de confirmaciones de cambio. Algunos de estos modos no son precisamente obvios, pero conviene conocerlos.

#### **Confirmaciones puntuales**

La forma canónica de referirse a una confirmación de cambios es indicando su código-resumen criptográfico SHA-1. Pero también existen otras maneras más sencillas. En esta sección se verán las diversas formas existentes para referirse a una determinada confirmación de cambios (commit).

#### **SHA corto**

Simplemente dándole los primeros caracteres del código SHA-1, Git es lo suficientemente inteligente como para figurarse cual es la confirmación de cambios (commit) deseada. Es

necesario teclear por lo menos 4 caracteres y estos han de ser no ambiguos --es decir, debe existir un solo objeto en el repositorio cuyo código comience por dicho trozo inicial del SHA--.

Por ejemplo, a la hora de localizar una confirmación de cambios, supongamos que se lanza el comando 'git log' e intentamos localizar la confirmación de cambios concreta donde se añadió una 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, escogiendo '1c002dd....', para lanzar el comando 'git show' sobre esa confirmación de cambios concreta, serían equivalentes todos estos comandos (asumiendo la no ambigüedad de todas las versiones cortas indicadas):

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

En todos estos casos, Git puede deducir el resto del valor SHA-1. Con la opción '--abbrev-commit' del comando 'git log', en su salida se mostrarán valores acortados, pero únicos de SHA.

Habitualmente suelen resultar valores de siete caracteres, pero alguno puede ser más largo si es necesario para preservar la unicidad de todos los valores SHA-1 mostrados:



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

Normalmente, entre ocho y diez caracteres suelen ser más que suficientes para garantizar la unicidad de todos los objetos dentro de cualquier proyecto. Aunque, en uno de los más grandes proyectos gestionados con Git, el kernel de Linux, están siendo necesarios unos 12 caracteres (de los 40 posibles) para garantizar la unicidad.

### Un breve comentario sobre los códigos SHA-1

Mucha gente se suele preocupar por si, por casualidad, dos objetos en su repositorio reciben el mismo código SHA-1 para identificarlos. ¿Y qué sucedería si se diera ese caso?

Si se da la casualidad de confirmar cambios en un objeto y que a este se le asigne el mismo código SHA-1 que otro ya existente en el repositorio. Al ver el objeto previamente almacenado en la base de datos, Git asumirá que este ya existía. Al intentar recuperar (check-out) el objeto más tarde, siempre se obtendrán los datos del primer objeto.

No obstante, hemos de ser conscientes de lo altamente improbable de un suceso así. Los códigos SHA-1 son de 20 bytes, (160 bits). El número de objetos, codificados aleatoriamente, necesarios para asegurar un 50% de probabilidad de darse una sola colisión es cercano a  $2^{80}$  (la fórmula para determinar la probabilidad de colisión es  $p = (n(n-1)/2) * (1/2^{160})$ ).  $2^{80}$  es  $1'2 \times 10^{24}$ , o lo que es lo mismo, 1 billón de billones. Es decir, unas 1.200 veces el número de granos de arena en la Tierra.

El siguiente ejemplo puede ser bastante ilustrativo, para hacernos una idea de lo que podría tardarse en darse una colisión en el código SHA-1: Si todos los 6'5 billones de humanos en el planeta Tierra estuvieran programando y, cada segundo, cada uno de ellos escribiera código equivalente a todo el histórico del kernel de Linux (cerca de 1 millón de objetos Git), enviándolo todo a un enorme repositorio Git. Serían necesarios unos 5 años antes de que dicho repositorio contuviera suficientes objetos como para tener una probabilidad del 50% de darse una sola colisión en el código SHA-1. Es mucho más probable que todos los miembros de nuestro equipo de programación fuesen atacados y matados por lobos, en incidentes no relacionados entre sí, acaecidos todos ellos en una misma noche.

### Referencias a ramas

La manera más directa de referirse a una confirmación de cambios es teniendo una rama apuntando a ella. De esta forma, se puede emplear el nombre de la rama en cualquier comando Git que espere un objeto de confirmación de cambios o un código SHA-1. Por ejemplo, si se desea mostrar la última confirmación de cambios en una rama, y suponiendo que la rama 'topic1' apunta a 'ca82a6d', los tres comandos siguientes son equivalentes:

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

Para ver a qué código SHA apunta una determinada rama, o si se desea conocer cómo se comportarían cualquiera de los ejemplos anteriores en términos de SHAs, se puede emplear el comando de fontanería 'rev-parse'. En el capítulo 9 se verá más información sobre las herramientas de fontanería. Herramientas estas que son utilizadas para operaciones a muy bajo nivel, y que no están pensadas para ser utilizadas en el trabajo habitual del día a día. Pero que, sin embargo, pueden ser muy útiles cuando se desea ver lo que realmente sucede "tras las bambalinas", en el interior de Git. Por ejemplo, lanzando el comando 'rev-parse' sobre una rama, esta muestra el código SHA-1 de la última confirmación de cambios en ella:

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

## **Nombres cortos en RefLog**

Una de las tareas realizadas por Git continuamente en segundo plano, mientras nosotros trabajamos, es el mantenimiento de un registro de referencia (reflog). En este registro queda traza de dónde han estado las referencias a HEAD y a las distintas ramas durante los últimos meses.

Este registro de referencia se puede consultar con el comando '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 se actualiza una rama por cualquier razón, Git almacena esa información en este histórico temporal. Y esta información se puede utilizar para referirse a confirmaciones de cambio pasadas. Por ejemplo, si se desea ver el quinto anterior valor de HEAD en el repositorio, se puede emplear la referencia '@{n}' mostrada por la salida de reflog:

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

Esta misma sintaxis puede emplearse cuando se desea ver dónde estaba una rama en un momento específico en el tiempo. Por ejemplo, para ver dónde apuntaba la rama 'master' en el día de ayer, se puede teclear:

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

Este comando mostrará a dónde apuntaba ayer la rama. Esta técnica tan solo funciona para información presente en el registro de referencia. No se puede emplear para confirmaciones de cambio de antigüedad superior a unos pocos meses.

Si se desea ver la información del registro de referencia, formateada de forma similar a la salida del comando 'git log', se puede lanzar el comando '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 destacar la estricta localidad de la información en el registro de referencia. Es un registro que se va componiendo en cada repositorio según se va trabajando en él. Las referencias

de una cierta persona en su repositorio nunca serán las mismas que las de cualquier otra persona en su copia local del repositorio. Es más, justo tras terminar de clonar un repositorio lo que se tiene es un registro de referencia vacío, puesto que aún no se ha realizado ningún trabajo sobre dicho repositorio recién clonado. Así, un comando tal como `git show HEAD@{2.months.ago}` solo será válido en caso de haber clonado el proyecto como mínimo dos meses antes. Si se acaba de clonar hace cinco minutos, ese comando dará un resultado vacío.

## Referencias a ancestros

Otra forma de especificar una confirmación de cambios es utilizando sus ancestros. Colocando un '^' al final de una referencia, Git interpreta que se refiere al padre de dicha referencia.

Suponiendo que sea esta 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
```

Se puede visualizar la anteúltima confirmación de cambios indicando 'HEAD^', 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 es posible indicar un número detrás de '^'. Por ejemplo `d921970^2`, para indicar "el segundo padre de d921970". Aunque esta sentencia es útil tan solo en confirmaciones de fusiones (merge), los únicos tipos de confirmación de cambios que pueden tener más de un padre. El primer padre es el proveniente de la rama activa al realizar la fusión, y el segundo es la confirmación de cambios en la rama desde la que se fusiona.

```
$ 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

Otra forma de referirse a los ancestros es la marca `~`. Utilizada tal cual, también se refiere al padre. Por lo tanto, `HEAD~` y `HEAD^` son equivalentes. Pero la diferencia comienza al indicar un número tras ella. `HEAD~2` significa "el primer padre del primer padre", es decir, "el abuelo". Y así según el número de veces que se indique. Por ejemplo, en la historia de proyecto citada anteriormente, `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

Igualmente, se podría haber escrito `HEAD^^^`, que también se refiere al "primer padre del primer padre del primer padre":

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

ignore \*.gem

E incluso también es posible combinar las dos sintaxis. Por ejemplo, para referirse al "segundo padre de la referencia previa" (asumiendo que es una confirmación de cambios de fusión -merge-), se puede escribir algo como `HEAD~3^2`.

## Referencias a un rango de confirmaciones de cambios

Una vez vistas las formas de referirse a confirmaciones concretas de cambios. Vamos a ver cómo referirse a un grupo de confirmaciones. Esto es especialmente útil en la gestión de ramas. Si se tienen multitud de ramas, se pueden emplear las especificaciones de rango para responder a cuestiones tales como "¿cual es el trabajo de esta rama que aún no se ha fusionado con la rama principal?".

### **Doble punto**

La especificación de rango más común es la sintaxis doble-punto. Básicamente, se trata de pedir a Git que resuelva un rango de confirmaciones de cambio alcanzables desde una confirmación determinada, pero no desde otra. Por ejemplo, teniendo un historial de confirmaciones de cambio tal como el de la figura 6-1.

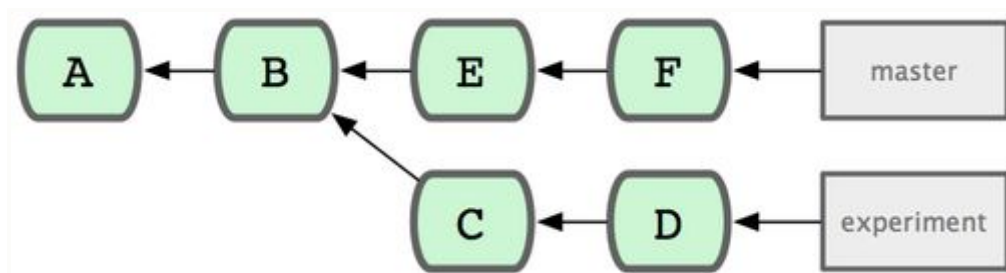


Figura 6-1. Ejemplo de historial para selección de rangos.

Si se desea ver qué partes de la rama experiment están sin fusionar aún con la rama master. Se puede pedir a Git que muestre un registro con las confirmaciones de cambio en `master..experiment`. Es decir, "todas las confirmaciones de cambio alcanzables desde experiment que no se pueden alcanzar desde master". Por razones de brevedad y claridad en los ejemplos, para representar los objetos confirmación de cambios (commit) se utilizarán las letras mostradas en el diagrama en lugar de todo el registro propiamente dicho:

```
$ git log master..experiment
```

```
D
```

```
C
```

Si, por el contrario, se desea ver lo opuesto (todas las confirmaciones en 'master' que no están en 'experiment'). Simplemente hay que invertir los nombres de las ramas. `experiment..master` muestra todo lo que haya en 'master' pero que no es alcanzable desde 'experiment':

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

Esto es útil si se desea mantener actualizada la rama 'experiment' y previsualizar lo que se está a punto de fusionar en ella. Otra utilidad habitual de estas sentencias es la de ver lo que se está a punto de enviar a un repositorio remoto:

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

Este comando muestra las confirmaciones de cambio de la rama activa que no están aún en la rama 'master' del repositorio remoto 'origin'. Si se lanza el comando 'git push' (y la rama activa actual esta relacionada con 'origin/master'), las confirmaciones de cambio mostradas por `git log origin/master..HEAD` serán las que serán transferidas al servidor. Es posible también omitir la parte final de la sentencia y dejar que Git asuma HEAD. Por ejemplo, se pueden obtener los mismos resultados tecleando `git log origin/master..`, ya que git sustituye HEAD en la parte faltante.

### Puntos multiples

La sintaxis del doble-punto es útil como atajo. Pero en algunas ocasiones interesa indicar más de dos ramas para precisar la revisión. Como cuando se desea ver las confirmaciones de cambio presentes en cualquiera de varias ramas y no en la rama activa. Git permite realizar esto utilizando o bien el carácter '^' o bien la opción `--not` por delante de aquellas referencias de las que se desea no ver las confirmaciones de cambio. Así, estos tres comandos son equivalentes:

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

Esto nos permite indicar más de dos referencias en una misma consulta. Algo imposible con la sintaxis dos-puntos. Por ejemplo, si se deseean ver todas las confirmaciones de cambio alcanzables desde la 'refA' o la 'refB', pero no desde la 'refC', se puede teclear algo como esto:

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

Esto da una enorme versatilidad al sistema de consultas y permite revisar el contenido de todas las ramas en el repositorio.

### Triple-punto

La última de las opciones principales para seleccionar rangos es la sintaxis triple-punto. Utilizada para especificar todas las confirmaciones de cambio alcanzables separadamente desde cualquiera de dos referencias, pero no desde ambas a la vez. Volviendo sobre la historia de proyecto mostrada en la figura 6-1. Si se desea ver lo que está o bien en 'master' o bien en 'experiment', pero no en ambas simultáneamente, se puede emplear el comando:

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

De nuevo, esto da una salida normal de 'log', pero mostrando tan solo información sobre las cuatro confirmaciones de cambio, dadas en la tradicional secuencia ordenada por fechas.

Una opción habitual a utilizar en estos casos con el comando 'log' suele ser 'left-right'. Haciendo así que en la salida se muestre cual es el lado al que pertenece cada una de las confirmaciones de cambio. Esto hace más útil la información mostrada:

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

Con estas herramientas, es mucho más sencillo indicar con precisión cual o cuales son las confirmaciones de cambios que se desean revisar.



## 6.2 Las herramientas de Git - Preparación interactiva

### Preparación interactiva

Git trae incluidos unos cuantos scripts para facilitar algunas de las tareas en la línea de comandos. Se van a mostrar unos pocos comandos interactivos que suelen ser de gran utilidad a la hora de recoger en una confirmación de cambios solo ciertas combinaciones y partes de archivos. Estas herramientas son útiles, por ejemplo, cuando se modifican unos cuantos archivos y luego se decide almacenar esos cambios en una serie de confirmaciones de cambio focalizadas en lugar de en una sola confirmación de cambio entremezclada. Así, se consiguen unas confirmaciones de cambio con agrupaciones lógicas de modificaciones, facilitando su revisión por parte otros desarrolladores que trabajen con nosotros. Al lanzar el comando 'git add' con las opciones '-i' o '--interactive', Git entra en un modo interactivo y muestra algo así como:

```
$ 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>
```

Según se ve, este comando muestra una vista bastante diferente del área de preparación (staging area). Básicamente se trata de la misma información dada por el comando 'git status', pero mas sucinta e informativa. Se ve una lista de cambios ya preparados, en la izquierda; y de los que están aún sin preparar, en la derecha.

Tras esa lista, viene la sección de comandos. Aquí se pueden lanzar acciones tales como: añadir archivos en el area de preparación (staging), sacar archivos de ella (unstaging), poner solo parte de algún archivo, añadir archivos nuevos que estaban fuera del sistema de control o mostrar diferencias en aquello que se ha añadido.

## Introduciendo archivos en el area de preparación y sacandolos de ella

Tecleando '2' o 'u' (update) tras el indicador 'What now>', el script interactivo preguntará cuales son los archivos que se quieren añadir al área de preparación:

```
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 añadir los archivos TODO e index.html, se teclearian 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 asterisco \* al lado de cada archivo indica que dicho archivo ha sido seleccionado para ser preparado. Pulsando la tecla [Enter] tras el indicador 'Update>>', Git toma lo seleccionado y lo añade al área de preparación:

```
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
```

En estos momentos se ve que los archivos TODO e index.html están en el área de preparación y que el archivo simplegit.rb no está aún. Si se desea sacar el archivo TODO del área, se puede utilizar la opción '3' o 'r' (revert):

\*\*\* 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

Volviendo a mirar el estado de Git, se comprueba que se ha sacado el archivo TODO del área de preparación:

\*\*\* 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 las diferencias entre lo que está preparado, se puede utilizar la opción '6' o 'd' (diff). Esta muestra una lista de los archivos preparados en el área de preparación, permitiendo la selección de aquellos sobre los que se desean ver diferencias. Es muy parecido a lanzar el comando 'git diff --cached' directamente en la línea de comandos:

\*\*\* 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, se ha visto cómo se puede emplear el modo interactivo para interactuar de forma más sencilla con el área de preparación.

### Parches en la preparación

También es posible añadir solo ciertas partes de algunos archivos y no otras. Por ejemplo, si se han realizado dos cambios en el archivo simplegit.rb y se desea pasar solo uno de ellos al área de preparación, pero no el otro. En el indicador interactivo se ha de teclear '5' o 'p' (patch). Git preguntará cual es el archivo a pasar parcialmente al área de preparación. Y después irá mostrando trozos de las distintas secciones modificadas en el archivo, preguntando por cada una si se desea pasar o no al área de preparación:

```
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,]?
```

En estas preguntas, hay varias opciones de respuesta. Tecleando '?' se muestra una lista de las mismas:

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

Habitualmente se tecleará 'y' o 'n' según se desee pasar o no cada trozo. Pero habrá ocasiones donde pueda ser útil pasar todos ellos conjuntamente, o el dejar para más tarde la decisión sobre un trozo concreto. Si se decide pasar solo una parte de un archivo y dejar sin pasar otra parte, la salida de estado mostrará algo así como:

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

La línea correspondiente al estado del archivo simplegit.rb es bastante interesante. Muestra que un par de líneas han sido preparadas (staged) en el área de preparación y otro par han sido dejadas fuera de dicho área (unstaged). Es decir, se ha pasado parcialmente ese archivo al área de preparación. En este punto, es posible salir del script interactivo y lanzar el comando 'git commit' para almacenar esa confirmación de cambios parciales en los archivos.

Por último, cabe comentar que no es necesario entrar expresamente en el modo interactivo para preparar archivos parcialmente. También se puede acceder a ese script con los comandos 'git add -p' o con 'git add --patch', directamente desde la línea de comandos.

## 6.3 Las herramientas de Git - Guardado rápido provisional

### Guardado rápido provisional

Según se está trabajando en un apartado de un proyecto, normalmente el espacio de trabajo suele estar en un estado inconsistente. Pero puede que se necesite cambiar de rama durante un breve tiempo para ponerse a trabajar en algún otro tema urgente. Esto plantea el problema de confirmar cambios en un trabajo medio hecho, simplemente para poder volver a ese punto más tarde. Y su solución es el comando 'git stash'.

Este comando de guardado rápido (stashing) toma el estado del espacio de trabajo, con todas las modificaciones en los archivos bajo control de cambios, y lo guarda en una pila provisional. Desde allí, se podrán recuperar posteriormente y volverlas a aplicar de nuevo sobre el espacio de trabajo.

### Guardando el trabajo temporalmente

Por ejemplo, si se está trabajando sobre un par de archivos e incluso uno de ellos está ya añadido al área de preparación para un futuro almacenamiento de sus cambios en el repositorio. Al lanzar el comando 'git status', se podría observar un estado inconsistente tal como:

```
$ git status
# On branch master
# 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)
#
#       modified:   lib/simplegit.rb
#
```

Si justo en este momento se desea cambiar de rama, pero sin confirmar los cambios realizados hasta entonces; la solución es un guardado rápido provisional de los cambios. Utilizando el comando 'git stash' y enviando un nuevo grupo de cambios a la pila de guardado rápido:

```
$ 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")
```

Con ello, se limpia el área de trabajo:

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

Y se permite cambiar de rama para ponerse a trabajar en cualquier otra parte. Con la tranquilidad de que los cambios a medio completar están guardados a buen recaudo en la pila de guardado rápido. Para ver el contenido de dicha pila, se emplea el comando '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 ejemplo, se habían realizado dos guardados rápidos anteriores, por lo que se ven tres grupos de cambios guardados en la pila. Con el comando 'git stash apply', tal y como se indica en la salida del comando stash original, se pueden volver a aplicar los últimos cambios recién guardados. Si lo que se desea es reaplicar alguno de los grupos más antiguos de cambios, se ha de indicar expresamente: **git stash apply stash@{2}** Si no se indica ningún grupo concreto, Git asume que se desea reaplicar el grupo de cambios más reciente de entre los guardados en la pila.

```
$ git stash apply
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#
#       modified:   index.html
#       modified:   lib/simplegit.rb
```

#

Como se ve en la salida del comando, Git vuelve a aplicar los correspondientes cambios en los archivos que estaban modificados. Pero no conserva la información de lo que estaba o no estaba añadido al área de preparación. En este ejemplo se han aplicado los cambios de vuelta sobre un espacio de trabajo limpio, en la misma rama. Pero no es esta la única situación en la que se pueden reaplicar cambios. Es perfectamente posible guardar rápidamente (stash) el estado de una rama. Cambiar posteriormente a otra rama. Y proceder a aplicar sobre esta otra rama los cambios guardados, en lugar de sobre la rama original. Es posible incluso aplicar de vuelta cambios sobre un espacio de trabajo inconsistente, donde haya otros cambios o algunos archivos añadidos al área de preparación. (Git notificará de los correspondientes conflictos de fusión si todo ello no se puede aplicar limpiamente.)

Las modificaciones sobre los archivos serán aplicadas; pero no así el estado de preparación. Para conseguir esto último, es necesario emplear la opción `--index` del comando `git stash apply`. Con ella se le indica que debe intentar reaplicar también el estado de preparación de los archivos. Y así se puede conseguir volver exactamente al punto original:

```
$ git stash apply --index
# On branch master
# 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)
#
#       modified:   lib/simplegit.rb
#
```

Los comandos `git stash apply` tan solo recuperan cambios almacenados en la pila de guardado rápido, sin afectar al estado de la pila. Es decir, los cambios siguen estando guardados en la pila. Para quitarlos de ahí, es necesario lanzar expresamente el comando `git stash drop` e indicar el número de guardado a borrar de la pila:

```
$ 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 es posible utilizar el comando `git stash pop`, que aplica cambios de un guardado y lo retira inmediatamente de la pila.

### Creando una rama desde un guardado rápido temporal

Si se almacena rápidamente (stash) un cierto trabajo, se deja en la pila durante bastante tiempo, y se continua mientras tanto con otros trabajos sobre la misma rama. Es muy posible que se presenten problemas al tratar de reaplicar los cambios guardados tiempo atrás. Si para recuperar esos cambios se ha de modificar un archivo que también haya sido modificado en los trabajos posteriores, se dará un conflicto de fusión (merge conflict) y será preciso resolverlo manualmente. Una forma más sencilla de reaplicar cambios es utilizando el comando `git stash branch`. Este comando crea una nueva rama, extrayendo (checkout) la confirmación de cambios original en la que se estaba cuando los cambios fueron guardados en la pila, replica estos sobre dicha rama y los borra de la pila si se consigue completar el proceso 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
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#
#       modified:   lib/simplegit.rb
#
Dropped refs/stash@{0} (f0dfc4d5dc332d1cee34a634182e168c4efc3359)
```

Este es un buen atajo para recuperar con facilidad un cierto trabajo desde la pila y continuar con él en una nueva rama.

## 6.4 Las herramientas de Git - Reescribiendo la historia

### Reescribiendo la historia

Por razones varias, hay ocasiones en que se desea revisar el historial de confirmaciones de cambio. Una de las grandes características de Git es su capacidad de postponer las decisiones hasta el último momento. Las decisiones sobre qué archivos van en qué confirmaciones de cambio se toman justo inmediatamente antes de confirmar, utilizando para ello el área de preparación (staging area). En cualquier momento se puede decidir dejar de trabajar en una cierta vía y arrancar en otra, utilizando el comando de guardado rápido (stash). Y también es posible reescribir confirmaciones de cambio ya realizadas, para que se muestren como si hubieran sido realizadas de otra forma. Así, es posible cambiar el orden de las confirmaciones, cambiar sus mensajes, modificar los archivos comprendidos en ellas, juntar varias confirmaciones en una sola, partir una en varias, o incluso borrar alguna completamente. --Aunque todo ello es siempre recomendable hacerlo solo antes de compartir nuestro trabajo con otros.--

En esta sección, se verá cómo realizar todas esas útiles tareas. De tal forma que se pueda dejar el historial de cambios exactamente tal y como se desee. Eso sí, siempre antes de compartirlo con otros desarrolladores.

### Modificar la última confirmación de cambios

Modificar la última confirmación de cambios (commit) es probablemente el arreglo realizado con más frecuencia. Dos suelen ser los cambios básicos a realizar: cambiar el mensaje o cambiar los archivos añadidos, modificados o borrados.

Cambiar el mensaje de la última confirmación de cambios, es muy sencillo:

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

Mediante este comando, el editor de textos arranca con el mensaje escrito en la última confirmación de cambios; listo para ser modificado. Al guardar y cerrar en el editor, este escribe una nueva confirmación de cambios y reemplaza con ella la última confirmación existente.

Si se desea cambiar la instantánea (snapshot) de archivos en la última confirmación de cambios, habitualmente por haber tenido algún descuido al añadir algún archivo de reciente creación. El proceso a seguir es básicamente el mismo. Se preparan en el área de preparación los archivos deseados; con los comandos `git add` o `git rm`, según corresponda. Y, a continuación, se lanza el comando `git commit --amend`. Este tendrá en cuenta dicha preparación para rehacer la instantánea de archivos en la nueva confirmación de cambios.

Es importante ser cuidadoso con esta técnica. Porque al modificar cualquier confirmación de cambios, cambia también su código SHA-1. Es como si se realizara una pequeña reorganización (rebase). Y, por tanto, aquí también se aplica la regla de no modificar nunca una confirmación de cambios que ya haya sido enviada (push) a otros.

### Modificar múltiples confirmaciones de cambios

Para modificar una confirmación de cambios situada bastante atrás en el historial, es necesario emplear herramientas más complejas. Git no dispone de herramientas directas para modificar el historial de confirmaciones de cambio. Pero es posible emplear la herramienta de reorganización (rebase) para modificar series de confirmaciones; en la propia cabeza (HEAD) donde estaban basadas originalmente, en lugar de moverlas a otra distinta. Dentro de la herramienta de reorganización interactiva, es posible detenerse justo tras cada confirmación de cambios a modificar. Para cambiar su mensaje, añadir archivos, o cualquier otra modificación. Este modo interactivo se activa utilizando la opción `-i` en el comando `git rebase`. La profundidad en la historia a modificar vendrá dada por la confirmación de cambios (commit) que se indique al comando.

Por ejemplo, para modificar las tres últimas confirmaciones de cambios, se indicara el padre de la última confirmación a modificar, es decir habrá que escribir `HEAD~2^` or `HEAD~3` tras el comando `git rebase -i`. La nomenclatura `~3` es la mas sencilla de recordar, porque lo que se desea es modificar las tres últimas confirmaciones. Pero sin perder de vista que realmente se está señalando a cuatro confirmaciones de cambio más atras, al padre de la última de las confirmaciones de cambio a modificar.

`$ git rebase -i HEAD~3`

Es importante avisar de nuevo que se trata de un comando de reorganización: todas y cada una de las confirmaciones de cambios en el rango `HEAD~3..HEAD` van a ser reescritas, (cambia su código SHA-1), tanto si se modifica algo en ellas como si no. Por tanto, es importante no afectar a ninguna confirmación de cambios que haya sido ya enviada (push) a un servidor central. So pena

de confundir a otros desarrolladores, a los cuales se estaría dando una versión alternativa de un mismo cambio.

Al lanzar este comando, se verán una lista de confirmaciones de cambio en la pantalla del editor de textos:

```
pick f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-filepick 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
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
#
# If you remove a line here THAT COMMIT WILL BE LOST.
# However, if you remove everything, the rebase will be aborted.
###
```

Es importante destacar que esas confirmaciones de cambios se han listado en el orden opuesto al que normalmente son mostradas en el comando **log**. En este último, se suele ver algo así como:

```
$ 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
```

Prestar atención al orden inverso. La reorganización interactiva lanza un script. Un script que, comenzando por la confirmación de cambios indicada en la línea del comando (**HEAD~3**), va a reaplicar los cambios introducidos en cada una de las confirmaciones, desde arriba hasta abajo. En la lista se ven las mas antiguas encima, en lugar de las más recientes, precisamente porque esas van a ser las primeras en reaplicarse.

Para que el script se detenga en cada confirmación de cambios a modificar, hay que editarlo. Y se ha de cambiar la palabra 'pick' por la palabra 'edit' en cada una de las confirmaciones de cambio

donde se desee detener el script. Por ejemplo, para modificar solo el mensaje de la tercera confirmación de cambios, el script quedaria:

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

Cuando se guarde y cierre en el editor, Git hará un rebobinado hacia atras hasta la última de las confirmaciones de cambios en la lista, y mostrará algo así como:

```
$ git rebase -i HEAD~3
Stopped at 7482e0d... updated the gemspec to hopefully work better
You can amend the commit now, with
```

```
git commit --amend
```

Once you're satisfied with your changes, run

```
git rebase --continue
```

Estas instrucciones indican exactamente lo que se ha de realizar. Teclear

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

Cambiar el mensaje de la confirmación de cambios y salir del editor. Para luego lanzar

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

Las otras dos confirmaciones de cambio serán reaplicadas automáticamente. Y ya estará completa la reorganización. Si se ha cambiado 'pick' por 'edit' en más de una línea, estos pasos se habrán de repetir por cada una de las confirmaciones de cambios a modificar. En cada una de ellas, Git se detendrá, permitiendo enmendar la confirmación de cambios y continuar tras la modificación.

### **Reordenar confirmaciones de cambios**

Las reorganizaciones interactivas también se pueden emplear para reordenar o para eliminar completamente ciertas confirmaciones de cambios (commits). Por ejemplo, si se desea eliminar la confirmación de "added cat-file" y cambiar el orden en que se han introducido las otras dos confirmaciones de cambios, el script de reorganización pasaría de ser:

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

a quedar en algo como:

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

Cuando se guarde y salga en el editor, Git rebobinará la rama hasta el padre de las confirmaciones de cambio indicadas, reaplicará **310154e** y luego **f7f3f6d**, para finalmente detenerse. De esta forma se habrá cambiado el orden de las dos confirmaciones de cambio, y se habrá eliminado completamente la de "added cat-file".

### **Combinar varias confirmaciones en una sola**

Con la herramienta de reorganización interactiva, es posible recombinar una serie de confirmaciones de cambio y agruparlas todas en una sola. El propio script indica las instrucciones a seguir:

```
#
# Commands:
# p, pick = use commit
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
#
# If you remove a line here THAT COMMIT WILL BE LOST.
# However, if you remove everything, the rebase will be aborted.
###
```

Si, en lugar de 'pick' o de 'edit', se indica 'squash' delante de alguna de las confirmaciones de cambio, Git aplicará simultáneamente dicha confirmación y la que esté inmediatamente delante de ella. Permitiendo también combinar los mensajes de ambas. Por ejemplo, si se desea hacer una única confirmación de cambios fusionando las tres, el script quedaría en algo como:

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

squash a5f4a0d added cat-file

Cuando se guarde y salga en el editor, Git rebobinará la historia, reaplicará las tres confirmaciones de cambio, y volverá al editor para fusionar también los mensajes de esas tres confirmaciones.

# 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

Al guardar esto, se tendrá una sola confirmación de cambios que introducirá todos los cambios que estaban en las tres confirmaciones de cambios previamente existentes.

### Dividir una confirmación de cambios en varias

Dividir una confirmación de cambios (commit), implica deshacerla y luego volver a preparar y confirmar trozos de la misma tantas veces como nuevas confirmaciones se desean tener al final. Por ejemplo, si se desea dividir la confirmación de cambios de enmedio de entre las tres citadas en ejemplos anteriores. Es decir, si en lugar de "updated README formatting and added blame", se desea separar esa confirmación en dos: "updated README formatting" y "added blame". Se puede realizar cambiando la instrucción en el script de **rebase -i**, desde 'split' a 'edit':

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

Después, cuando el script devuelva la línea de comandos, se ha de deshacer (reset) esa confirmación de cambios, coger los cambios recién deshechos y crear múltiples nuevas confirmaciones de cambios con ellos. Al guardar y salir en el editor, Git rebobinará la historia hasta el padre de la primera confirmación de cambios en la lista, reaplicará esa primera confirmación (**f7f3f6d**), luego reaplicará la segunda (**310154e**) y luego devolverá la línea de

comandos. En esta línea de comando, es donde se deshacen los cambios tecleando el comando **git reset HEAD^** para dejar sin preparar (unstaged) los archivos cambiados. Para, seguidamente, elaborar tantas confirmaciones de cambios como se desee, a base de pasar archivos al área de preparación y confirmarlos. Y, finalmente, teclear el comando **git rebase --continue** para completar la tarea.

```
$ 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
```

Tras esto, Git reaplicará la última de las confirmaciones de cambios (**a5f4a0d**) en el script. Quedando la historia:

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

De nuevo, merece recalcar el hecho de que estas operaciones cambian los códigos SHA-1 de todas las confirmaciones de cambio afectadas. Y que, por tanto, no se deben hacer sobre confirmaciones de cambio enviadas(push) a algún repositorio compartido.

### **La opción nuclear: filter-branch**

Existe una opción de reescritura del historial que se puede utilizar si se necesita reescribir un gran número de confirmaciones de cambio de forma mas o menos automatizada. Por ejemplo, para cambiar una dirección de correo electrónico globalmente, o para quitar un archivo de todas y cada una de las confirmaciones de cambios en una determinada rama. El comando en cuestión es **filter-branch**, y permite reescribir automáticamente grandes porciones del historial.

Precisamente por ello, no debería utilizarse a no ser que el proyecto aún no se haya hecho público (es decir, otras personas no han basado su trabajo en alguna de las confirmaciones de cambio que se van a modificar). De todas formas, allá donde sea aplicable, puede ser de gran utilidad. Se van a ilustrar unas cuantas de las ocasiones donde se podría utilizar, para dar así una idea de sus capacidades.



## Quitar un archivo de cada confirmación de cambios

Es algo que frecuentemente suele ser necesario. Alguien confirma cambios y almacena accidentalmente un enorme archivo binario cuando lanza un `git add` . sin pensarlo demasiado. Y es necesario quitarlo del repositorio. O podría suceder que se haya confirmado y almacenado accidentalmente un archivo que contiene una contraseña importante, Y el proyecto se va a hacer de código abierto. En estos casos, la mejor opción es utilizar la herramienta `filter-branch` para limpiar todo el historial. Por ejemplo, para quitar un archivo llamado `passwords.txt` del repositorio, se puede emplear la opción `--tree-filter` del comando `filter-branch`:

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

Esta opción `--tree-filter`, tras cada extracción (checkout) del proyecto, lanzará el comando especificado y reconfirmará los cambios resultantes(recommit). En esta ocasión, se eliminará un archivo llamado `passwords.txt` de todas y cada una de las instantáneas (snapshot) almacenadas, tanto si este existe como si no. Otro ejemplo: si se desean eliminar todos los archivos de respaldo del editor que han sido almacenados por error, se podría lanzar algo así como `git filter-branch --tree-filter "find * -type f -name '*~' -delete" HEAD`.

Y se iría viendo como Git reescribe árboles y confirmaciones de cambio, hasta que el apuntador de la rama llegue al final. Una recomendación: en general, suele ser buena idea lanzar cualquiera de estas operaciones primero sobre una rama de pruebas y luego reinicializar (hard-reset) la rama maestra (master), una vez se haya comprobado que el resultado de las operaciones es el esperado. Si se desea lanzar `filter-branch` sobre todas las ramas del repositorio, se ha de pasar la opción `--all` al comando.

## Haciendo que una subcarpeta sea la nueva carpeta raíz

Por ejemplo, en el caso de que se haya importado trabajo desde otro sistema de control de versiones, y se tengan algunas subcarpetas sin sentido (trunk, tags,...). `filter-branch` puede ser de utilidad para que, por ejemplo, la subcarpeta `trunk` sea la nueva carpeta raíz del proyecto en todas y cada una de las confirmaciones de cambios:

```
$ git filter-branch --subdirectory-filter trunk HEAD
Rewrite 856f0bf61e41a27326cdae8f09fe708d679f596f (12/12)
Ref 'refs/heads/master' was rewritten
```

Tras este comando, la nueva raíz del proyecto pasa a ser el contenido de la carpeta **trunk**. Y, además, Git elimina automáticamente todas las confirmaciones de cambio (commits) que no afectaban a dicha subcarpeta.

### **Cambiando direcciones de correo-e de forma global**

Otra utilidad típica para utilizar **filter-branch** es cuando alguien ha olvidado ejecutar **git config** para configurar su nombre y dirección de correo electrónico antes de comenzar a trabajar. O cuando se va a pasar a código abierto un proyecto, pero previamente se desea cambiar todas las direcciones de correo empresariales por direcciones personales. En cualquier caso, se pueden cambiar de un golpe las direcciones de correo en múltiples confirmaciones de cambio. Aunque es necesario ser cuidadoso para actuar solo sobre aquellas direcciones que se deseen cambiar, utilizando para ello la opción **--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
```

Este comando pasa por todo el repositorio y reescribe cada confirmación de cambios donde detecte la dirección de correo indicada, para reemplazarla por la nueva. Y, debido a que cada confirmación de cambios contiene el código SHA-1 de sus ancestros, este comando cambia también todos los códigos SHA del historial; no solamente los de las confirmaciones de cambio que contenían la dirección indicada.

## **6.5 Las herramientas de Git - Depuración con Git**

### **Depuración con Git**

Git dispone también de un par de herramientas muy útiles para tareas de depuración en los proyectos. Precisamente por estar Git diseñado para trabajar con casi cualquier tipo de proyecto, sus herramientas son bastante genéricas. Pero suelen ser de inestimable ayuda para cazar errores o las causas de los mismos cuando se detecta que algo va mal.

## Anotaciones en los archivos

Cuando se está rastreando un error dentro del código buscando localizar cuándo se introdujo y por qué, el mejor auxiliar para hacerlo es la anotación de archivos. Esta suele mostrar la confirmación de cambios (commit) que modificó por última vez cada una de las líneas en cualquiera de los archivos. Así, cuando se está frente a una porción de código con problemas, se puede emplear el comando `git blame` para anotar ese archivo y ver así cuándo y por quién fue editada por última vez cada una de sus líneas. En este ejemplo, se ha utilizado la opción `-L` para limitar la salida a las líneas desde la 12 hasta la 22:

```
$ git blame -L 12,22 simplegit.rb
^4832fe2 (Scott Chacon 2008-03-15 10:31:28 -0700 12) def show(tree =
'master')
^4832fe2 (Scott Chacon 2008-03-15 10:31:28 -0700 13)   command("git show
#{tree}")
^4832fe2 (Scott Chacon 2008-03-15 10:31:28 -0700 14) end
^4832fe2 (Scott Chacon 2008-03-15 10:31:28 -0700 15)
9f6560e4 (Scott Chacon 2008-03-17 21:52:20 -0700 16) def log(tree = 'master')
79eaf55d (Scott Chacon 2008-04-06 10:15:08 -0700 17)   command("git log
#{tree}")
9f6560e4 (Scott Chacon 2008-03-17 21:52:20 -0700 18) end
9f6560e4 (Scott Chacon 2008-03-17 21:52:20 -0700 19)
42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 20) def blame(path)
42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 21)   command("git blame
#{path}")
42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 22) end
```

Merece destacar que el primer campo mostrado en cada línea es el código SHA-1 parcial de la confirmación de cambios en que se modificó dicha línea por última vez. Los dos siguientes campos son sendos valores extraídos de dicha confirmación de cambios --el nombre del autor y la fecha--, mostrando quien y cuándo modificó esa línea. Detrás, vienen el número de línea y el contenido de la línea propiamente dicha. En el caso de las líneas con la confirmación de cambios `^4832fe2`, merece comentar que son aquellas presentes en el archivo cuando se hizo la confirmación de cambios original; (la confirmación en la que este archivo se incluyó en el

proyecto por primera vez). No habiendo sufrido esas líneas ninguna modificación desde entonces. Puede ser un poco confuso, debido a que la marca ^ se utiliza también con otros significados diferentes dentro de Git. Pero este es el sentido en que se utiliza aquí: para señalar la confirmación de cambios original.

Otro aspecto interesante de Git es la ausencia de un seguimiento explícito de archivos renombrados. Git simplemente se limita a almacenar instantáneas (snapshots) de los archivos, para después intentar deducir cuáles han podido ser renombrados. Esto permite preguntar a Git acerca de todo tipo de movimientos en el código. Indicando la opción **-C** en el comando **git blame**, Git analizará el archivo que se está anotando para intentar averiguar si alguno de sus fragmentos pudiera provenir de, o haber sido copiado de, algún otro archivo. Por ejemplo, si se estaba refactorizando un archivo llamado **GITServerHandler.m**, para trocearlo en múltiples archivos, siendo uno de estos **GITPackUpload.m**. Aplicando la opción **-C** de **git blame** sobre **GITPackUpload.m**, es posible ver de donde proviene cada sección del código:

```
$ git blame -C -L 141,153 GITPackUpload.m
f344f58d GITServerHandler.m (Scott 2009-01-04 141)
f344f58d GITServerHandler.m (Scott 2009-01-04 142) - (void)
gatherObjectShasFromC
f344f58d GITServerHandler.m (Scott 2009-01-04 143) {
70befddd GITServerHandler.m (Scott 2009-03-22 144)    //NSLog(@"GATHER
COMMI
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 145)
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 146)    NSString *parentSha;
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 147)    GITCommit *commit =
[g
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 148)
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 149)    //NSLog(@"GATHER
COMMI
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 150)
56ef2caf GITServerHandler.m (Scott 2009-01-05 151)    if(commit) {
56ef2caf GITServerHandler.m (Scott 2009-01-05 152)        [refDict setOb
56ef2caf GITServerHandler.m (Scott 2009-01-05 153)
```

Lo cual es realmente útil. Habitualmente suele mostrarse como confirmación de cambios original aquella confirmación de cambios desde la que se copió el código. Por ser esa la primera ocasión en que se han modificado las líneas en ese archivo. Git suele indicar la confirmación de cambios

original donde se escribieron las líneas, incluso si estas fueron escritas originalmente en otro archivo.

## **Búsqueda binaria**

La anotación de archivos es útil si se conoce aproximadamente el punto dónde se localizan los problemas. Pero no siendo ese el caso, y habiéndose realizado docenas o cientos de confirmaciones de cambio desde el último estado estable conocido, puede ser de utilidad el comando **git bisect**. Este comando **bisect** realiza una búsqueda binaria por todo el historial de confirmaciones de cambio, para intentar localizar lo más rápido posible aquella confirmación de cambios en la que se pudieron introducir los problemas.

Por ejemplo, en caso de aparecer problemas justo tras enviar a producción un cierto código que parecía funcionar bien en el entorno de desarrollo. Si, volviendo atras, resulta que se consigue reproducir el problema, pero cuesta identificar su causa. Se puede ir biseccionando el código para intentar localizar el punto del historial desde donde se presenta el problema. Primero se lanza el comando **git bisect start** para iniciar el proceso de búsqueda. Luego, con el comando **git bisect bad**, se le indica al sistema cual es la confirmación de cambios a partir de donde se han detectado los problemas. Y después, con el comando **git bisect good [good\_commit]**, se le indica cual es la última confirmación de cambios conocida donde el código funcionaba bien:

```
$ git bisect start
$ git bisect bad
$ git bisect good v1.0
Bisecting: 6 revisions left to test after this
[ecb6e1bc347ccecc5f9350d878ce677feb13d3b2] error handling on repo
```

Git averigua que se han dado 12 confirmaciones de cambio entre la confirmación marcada como buena y la marcada como mala. Y extrae la confirmación central de la serie, para comenzar las comprobaciones a partir de ahí. En este punto, se pueden lanzar las pruebas pertinentes para ver si el problema existe en esa confirmación de cambios extraída. Si este es el caso, el problema se introdujo en algún punto anterior a esta confirmación de cambios intermedia. Si no, el problema se introdujo en un punto posterior. Por ejemplo, si resultara que no se detecta el problema aquí, se indicaría esta circunstancia a Git tecleando **git bisect good**; para continuar la búsqueda:

```
$ git bisect good
Bisecting: 3 revisions left to test after this
[b047b02ea83310a70fd603dc8cd7a6cd13d15c04] secure this thing
```

Git extraerá otra confirmación de cambios, aquella a medio camino entre la que se acaba de chequear y la que se había indicado como errónea al principio. De nuevo, se pueden lanzar las pruebas para ver si el problema existe o no en ese punto. Si, por ejemplo, si existiera se indicaría ese hecho a Git tecleando **git bisect bad**:

```
$ git bisect bad
Bisecting: 1 revisions left to test after this
[f71ce38690acf49c1f3c9bea38e09d82a5ce6014] drop exceptions table
```

Con esto el proceso de búsqueda se completa y Git tiene la información necesaria para determinar dónde comenzaron los problemas. Git reporta el código SHA-1 de la primera confirmación de cambios problemática y muestra una parte de la información relativa a esta y a los archivos modificados en ella. Así podemos irnos haciendo una idea de lo que ha podido suceder para que se haya introducido un error en el código:

```
$ git bisect good
b047b02ea83310a70fd603dc8cd7a6cd13d15c04 is first bad commit
commit b047b02ea83310a70fd603dc8cd7a6cd13d15c04
Author: PJ Hyett <pjhyett@example.com>
Date: Tue Jan 27 14:48:32 2009 -0800
```

```
    secure this thing
```

```
:040000 040000 40ee3e7821b895e52c1695092db9bdc4c61d1730
f24d3c6ebcf639b1a3814550e62d60b8e68a8e4 M config
```

Al terminar la revisión, es obligatorio teclear el comando **git bisect reset** para devolver HEAD al punto donde estaba antes de comenzar todo el proceso de búsqueda. So pena de dejar el sistema en un estado inconsistente.

```
$ git bisect reset
```

Esta es una poderosa herramienta que permite chequear en minutos cientos de confirmaciones de cambio, para determinar rápidamente en que punto se pudo introducir el error. De hecho, si se dispone de un script que dé una salida o si el proyecto funciona correctamente y distinto de o si el proyecto tiene errores, todo este proceso de búsqueda con **git bisect** se puede automatizar completamente. Primero, como siempre, se indica el alcance de la búsqueda indicando las aquellas confirmaciones de cambio conocidas donde el proyecto estaba mal y donde estaba bien.

Se puede hacer en un solo paso. Indicando ambas confirmaciones de cambios al comando **bisect start**, primero la mala y luego la buena:

```
$ git bisect start HEAD v1.0  
$ git bisect run test-error.sh
```

De esta forma, se irá ejecutando automáticamente **test-error.sh** en cada confirmación de cambios que se vaya extrayendo. Hasta que Git encuentre la primera donde se presenten problemas. También se puede emplear algo como **make** o como **make tests** o cualquier otro método que se tenga para lanzar pruebas automatizadas sobre el sistema.

## 6.6 Las herramientas de Git - Submódulos

### Submódulos

Suele ser frecuente encontrarse con la necesidad de utilizar otro proyecto desde dentro del que se está trabajando. En ocasiones como, por ejemplo, cuando se utiliza una biblioteca de terceros, o cuando se está desarrollando una biblioteca independiente para ser utilizada en múltiples proyectos. La preocupación típica en estos escenarios suele ser la de cómo conseguir tratar ambos proyectos separadamente. Pero conservando la habilidad de utilizar uno dentro del otro.

Un ejemplo concreto. Supongamos que se está desarrollando un site web y creando feeds Atom. En lugar de escribir código propio para generar los feeds Atom, se decide emplear una biblioteca ya existente. Y dicha biblioteca se incluye desde una biblioteca compartida tal como CPAN install o Ruby gem; o copiando directamente su código fuente en el árbol del propio proyecto. La problemática en el primer caso radica en la dificultad de personalizar la biblioteca compartida. Y en la dificultad para su despliegue; ya que es necesario que todos y cada uno de los clientes dispongan de ella. La problemática en el segundo caso radica en las complicaciones para fusionar las personalizaciones realizadas por nosotros con futuras copias de la biblioteca original.

Git resuelve estas problemáticas utilizando submódulos. Los submódulos permiten mantener un repositorio Git como una subcarpeta de otro repositorio Git. Esto permite clonar un segundo repositorio dentro del repositorio del proyecto en que se está trabajando, manteniendo separadamente las confirmaciones de cambios en ambos repositorios.

## Trabajando con submódulos

Suponiendo, por ejemplo, que se desea añadir la biblioteca Rack (un interface Ruby de pasarela de servidor web) al proyecto en que se está trabajando. Posiblemente con algunas personalizaciones, pero sin perder la capacidad de fusionar nuestros cambios con la evolución de la biblioteca original. La primera tarea a realizar es clonar el repositorio externo dentro de una subcarpeta dentro del proyecto. Los proyectos externos se pueden incluir como submódulos mediante el comando `git submodule add`:

```
$ git submodule add git://github.com/chneukirchen/rack.git rack
Initialized empty Git repository in /opt/subtest/rack/.git/
remote: Counting objects: 3181, done.
remote: Compressing objects: 100% (1534/1534), done. remote: Compressing
objects: 100% (1534/1534), done.
remote: Total 3181 (delta 1951), reused 2623 (delta 1603)
Receiving objects: 100% (3181/3181), 675.42 KiB | 422 KiB/s, done.
Resolving deltas: 100% (1951/1951), done. Resolving deltas: 100% (1951/1951),
done.
```

A partir de este momento, el proyecto Rack está dentro de nuestro proyecto; bajo una subcarpeta denominada `rack`. En dicha subcarpeta es posible realizar cambios, añadir un repositorio propio a donde enviar (push) los cambios, recuperar (fetch) y fusionar (merge) desde el repositorio original, y mucho mas... Si se lanza `git status` nada mas añadir el submódulo, se aprecian dos cosas:

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

Una: el archivo `.gitmodules`. un archivo de configuración para almacenar las relaciones entre la URL del proyecto y la subcarpeta local donde se ha colocado este.

```
$ cat .gitmodules
[submodule "rack"]
    path = rack
```



```
url = git://github.com/chneukirchen/rack.git
```

En caso de haber múltiples submódulos, habrá múltiples entradas en este archivo. Merece destacar que este archivo está también bajo el control de versiones, como lo están otros archivos tal como `.gitignore`, por ejemplo. Y será enviado (push) y recibido (pull) junto con el resto del proyecto. Así es como otras personas que clonen el proyecto pueden saber dónde encontrar los submódulos del mismo.

Dos: la entrada `rack`. Si se lanza un `git diff` sobre ella, se puede apreciar algo muy interesante:

```
$ git diff --cached rack
diff --git a/rack b/rack
new file mode 160000
index 0000000..08d709f
--- /dev/null
+++ b/rack
@@ -0,0 +1 @@
+Subproject commit 08d709f78b8c5b0fbeb7821e37fa53e69afcf433
```

Aunque `rack` es una subcarpeta de la carpeta de trabajo, git la contempla como un submódulo y no realiza seguimiento de sus contenidos si no se está situado directamente sobre ella. En su lugar, Git realiza confirmaciones de cambio particulares en ese repositorio. Cuando se realizan y confirman cambios en esa subcarpeta, el proyecto padre detecta el cambio en HEAD y almacena la confirmación de cambios concreta en la que se esté trabajando en ese momento. De esta forma, cuando otras personas clonen este proyecto, sabrán cómo recrear exactamente el entorno.

Esto es importante al trabajar con submódulos: siempre son almacenados como la confirmación de cambios concreta en la que están. No es posible almacenar un submódulo en `master` o en cualquier otra referencia simbólica.

Cuando se realiza una confirmación de cambios, se suele ver algo así como:

```
$ git commit -m 'first commit with submodule rack'
[master 0550271] first commit with submodule rack
2 files changed, 4 insertions(+), 0 deletions(-)
create mode 100644 .gitmodules
create mode 160000 rack
```

Notese el modo 160000 para la entrada **rack**. Este es un modo especial de Git, un modo en el que la confirmación de cambio se almacena como una carpeta en lugar de como una subcarpeta o un archivo.

Se puede considerar la carpeta **rack** como si fuera un proyecto separado. Y, como tal, de vez en cuando se puede actualizar el proyecto padre con un puntero a la última confirmación de cambios en dicho subproyecto. Todos los comandos Git actúan independientemente en ambas carpetas:

```
$ git log -1
commit 0550271328a0038865aad6331e620cd7238601bb
Author: Scott Chacon <schacon@gmail.com>
Date: Thu Apr 9 09:03:56 2009 -0700
```

```
    first commit with submodule rack
$ cd rack/
$ git log -1
commit 08d709f78b8c5b0fbef7821e37fa53e69afcf433
Author: Christian Neukirchen <chneukirchen@gmail.com>
Date: Wed Mar 25 14:49:04 2009 +0100
```

Document version change

## Clonando un proyecto con submódulos

Si se tiene un proyecto con submódulos dentro de él. Cuando se recibe, se reciben también las carpetas que contienen los submódulos; pero no se reciben ninguno de los archivos de dichos submódulos:

```
$ git clone git://github.com/schacon/myproject.git
Initialized empty Git repository in /opt/myproject/.git/
remote: Counting objects: 6, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 6 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (6/6), done.
$ cd myproject
$ ls -l
total 8
-rw-r--r-- 1 schacon admin  3 Apr  9 09:11 README
drwxr-xr-x 2 schacon admin 68 Apr  9 09:11 rack
$ ls rack/
```

\$

La carpeta **rack** está presente, pero vacía. Son necesarios otros dos comandos: **git submodule init** para inicializar el archivo de configuración local, y **git submodule update** para recuperar (fetch) todos los datos del proyecto y extraer (checkout) la confirmación de cambios adecuada desde el proyecto padre:

```
$ git submodule init
Submodule 'rack' (git://github.com/chneukirchen/rack.git) registered for path 'rack'
$ git submodule update
Initialized empty Git repository in /opt/myproject/rack/.git/
remote: Counting objects: 3181, done.
remote: Compressing objects: 100% (1534/1534), done. remote: Compressing
objects: 100% (1534/1534), done.
remote: Total 3181 (delta 1951), reused 2623 (delta 1603)
Receiving objects: 100% (3181/3181), 675.42 KiB | 173 KiB/s, done.
Resolving deltas: 100% (1951/1951), done. Resolving deltas: 100% (1951/1951),
done.
Submodule path 'rack': checked out '08d709f78b8c5b0fbef7821e37fa53e69afcf433'
```

Tras esto, la carpeta **rack** sí que está exactamente en el estado que le corresponde estar tras la última confirmación de cambios que se realizó sobre ella. Si otra persona realiza cambios en el código de **rack**, los confirma y nosotros recuperamos (pull) dicha referencia y la fusionamos (merge), se obtendrá un resultado un tanto extraño:

```
$ git merge origin/master
Updating 0550271..85a3eee
Fast forward
 rack | 2 +-
 1 files changed, 1 insertions(+), 1 deletions(-)
[master*]$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#    modified:   rack
#
```

Se ha fusionado en algo que es básicamente un cambio en el puntero al submódulo. Pero no se ha actualizado el código en la carpeta del submódulo propiamente dicha. Por lo que se muestra un estado inconsistente en la misma:

```
$ git diff
diff --git a/rack b/rack
index 6c5e70b..08d709f 160000
--- a/rack
+++ b/rack
@@ -1,1 @@
-Subproject commit 6c5e70b984a60b3cecd395edd5b48a7575bf58e0
+Subproject commit 08d709f78b8c5b0fbeb7821e37fa53e69afcf433
```

Siendo esto debido a que el puntero al submódulo que se tiene en este momento no corresponde a lo que realmente hay en carpeta del submódulo. Para arreglarlo, es necesario lanzar de nuevo el comando `git submodule update`:

```
$ git submodule update
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 1), reused 2 (delta 0)
Unpacking objects: 100% (3/3), done.
From git@github.com:schacon/rack
 08d709f..6c5e70b master -> origin/master
Submodule path 'rack': checked out
'6c5e70b984a60b3cecd395edd5b48a7575bf58e0'
```

Se necesita realizar este paso cada vez que se recupere (pull) un cambio del submódulo en el proyecto padre. Es algo extraño, pero ¡funciona!.

Un problema típico se suele dar cuando un desarrollador realiza y confirma (commit) un cambio local en el submódulo, pero no lo envía (push) a un servidor público. Pero, sin embargo, sí que confirma (commit) y envía (push) un puntero a dicho estado dentro del proyecto padre. Cuando otros desarrolladores intenten lanzar un `git submodule update`, será imposible encontrar la confirmación de cambios a la que se refiere el submódulo, ya que esta tan solo existe en el sistema del desarrollador original. En estos casos, se suele ver un error tal como:

```
$ git submodule update
fatal: reference isn't a tree: 6c5e70b984a60b3cecd395edd5b48a7575bf58e0
Unable to checkout '6c5e70b984a60b3cecd395edd5ba7575bf58e0' in submodule
```

```
path 'rack'
```

Forzandonos a mirar quién ha sido la persona que ha realizado los últimos cambios en el submódulo:

```
$ git log -1 rack
commit 85a3eee996800fcfa91e2119372dd4172bf76678
Author: Scott Chacon <schacon@gmail.com>
Date: Thu Apr 9 09:19:14 2009 -0700
```

added a submodule reference I will never make public. hahahahaha!

Para enviarle un correo-e y avisarle de su despiste.

## Proyectos padre

Algunas veces, dependiendo del equipo de trabajo en que se encuentren, los desarrolladores suelen necesitar mantener una combinación de grandes carpetas de proyecto. Se da frecuentemente en equipos procedentes de CVS o de Subversion (donde se define una colección de módulos o carpetas), cuando desean mantener ese mismo tipo de flujo de trabajo.

La manera más apropiada de hacer esto en Git, es la de crear diferentes repositorios, cada uno en su carpeta; para luego crear un repositorio padre que englobe múltiples submódulos, uno por cada carpeta. Un beneficio que se obtiene de esta manera de trabajar es la mayor especificidad en las relaciones entre proyectos, definidas mediante etiquetas (tag) y ramas (branch) en el proyecto padre.

## Posibles problemáticas al usar submódulos

El uso de submódulos tiene también sus contratiempos. El primero de los cuales es la necesidad de ser bastante cuidadoso cuando se trabaja en la carpeta de un submódulo. Al lanzar **git submodule update**, este comando comprueba la versión específica del proyecto, pero sin tener en cuenta la rama. Es lo que se conoce como "trabajar con cabecera desconectada" --es decir, el archivo HEAD apunta directamente a una confirmación de cambios (commit), y no a una referencia simbólica--. Este método de trabajo suele tenderse a evitar, ya que trabajando en un entorno de cabecera desconectada es bastante fácil despistarse y perder cambios ya realizados. Si se realiza un **submodule update** inicial, se hacen cambios y se confirman en esa carpeta de submódulo sin haber creado antes una rama en la que trabajar. Y si, tras esto, se realiza de nuevo un **git submodule update** desde el proyecto padre, sin haber confirmado cambios en este, Git

sobreescribirá cambios sin aviso previo. Técnicamente, no se pierde nada del trabajo.

Simplemente, nos quedamos sin ninguna rama apuntando a él. Con lo que resulta problemático recuperar el acceso a los cambios.

Para evitarlo, siempre se ha de crear una rama cuando se trabaje en la carpeta de un submódulo; usando `git checkout -b trabajo` o algo similar. Cuando se realice una actualización (update) del submódulo por segunda vez, se seguirá sobreescribiendo el trabajo; pero al menos se tendrá un apuntador para volver hasta los cambios realizados.

Intercambiar ramas con submódulos tiene también sus peculiaridades. Si se crea una rama, se añade un submódulo en ella y luego se retorna a una rama donde dicho submódulo no exista. La carpeta del submódulo sigue existiendo, solo que ahora queda como una carpeta sin seguimiento.

```
$ git checkout -b rack
Switched to a new branch "rack"
$ git submodule add git@github.com:schacon/rack.git rack
Initialized empty Git repository in /opt/myproj/rack/.git/
...
Receiving objects: 100% (3184/3184), 677.42 KiB | 34 KiB/s, done.
Resolving deltas: 100% (1952/1952), done.Resolving deltas: 100% (1952/1952),
done.Resolving deltas: 100% (1952/1952), done.
$ git commit -am 'added rack submodule'
[rack cc49a69] added rack submodule
 2 files changed, 4 insertions(+), 0 deletions(-)
 create mode 100644 .gitmodules
 create mode 160000 rack
$ git checkout master
Switched to branch "master"
$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       rack/
```

Forzandonos a removerla del camino. Lo cual obliga a volver a clonarla cuando se retome la rama inicial --con la consiguiente pérdida de los cambios locales si estos no habian sido enviados previamente al servidor--.

Y una última problemática en que se suelen encontrar quienes intercambian de carpetas a submódulos. Si se ha estado trabajando en archivos de un proyecto al que luego se desea convertir en un submódulo, hay que ser muy cuidadoso o Git se resentirá. Asumiendo que se tenían archivos en una carpeta 'rack' del proyecto, y que se desea intercambiarla por un submódulo. Si se borra la carpeta y luego se lanza un comando `submodule add`, Git avisará de "carpeta ya existente en el índice":

```
$ rm -Rf rack/  
$ git submodule add git@github.com:schacon/rack.git rack  
'rack' already exists in the index
```

Para evitarlo, se debe sacar la carpeta 'rack' del área de preparación. Después, Git permitirá la adicción del submódulo sin problemas:

```
$ git rm -r rack  
$ git submodule add git@github.com:schacon/rack.git rack  
Initialized empty Git repository in /opt/testsub/rack/.git/  
remote: Counting objects: 3184, done.  
remote: Compressing objects: 100% (1465/1465), 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 | 88 KiB/s, done.  
Resolving deltas: 100% (1952/1952), done.Resolving deltas: 100% (1952/1952),  
done.Resolving deltas: 100% (1952/1952), done.
```

Tras esto, y suponiendo que ese paso ha sido realizado en una rama. Si se intenta retornar a dicha rama, cuyos archivos están aún en el árbol actual en lugar de en el submódulo, se obtendrá el siguiente error:

```
$ git checkout master  
error: Untracked working tree file 'rack/AUTHORS' would be overwritten by merge.
```

Antes de cambiar a cualquier rama que no lo contenga, es necesario quitar de enmedio la carpeta del submódulo 'rack'.

```
$ mv rack /tmp/  
$ git checkout master  
Switched to branch "master"  
$ ls
```

Y, cuando se retorne a la rama anterior, se tendrá una carpeta 'rack' vacía. Ante lo cual, será necesario lanzar `git submodule update` para volver a clonarla; o, si no, volver a restaurar la carpeta `/tmp/rack` de vuelta sobre la carpeta vacía.

## 6.7 Las herramientas de Git - Fusión de subárboles

### Fusión de subárboles

Ahora que se han visto las dificultades que se pueden presentar utilizando el sistema de submódulos, es momento de echar un vistazo a una vía alternativa de atacar esa misma problemática. Cuando Git realiza una fusión, suele revisar lo que ha de fusionar entre sí y, tras ese análisis, elige la estrategia más adecuada para hacerlo. Si se están fusionando dos ramas, Git suele utilizar la *estrategia recursiva* (*recursive strategy*). Si se están fusionando más de dos ramas, Git suele escoger la *estrategia del pulpo* (*octopus strategy*). Estas son las estrategias escogidas por defecto, ya que la estrategia recursiva puede manejar complejas fusiones-de-tres-vías --por ejemplo, con más de un antecesor común-- pero tan solo puede fusionar dos ramas. La fusión-tipo-pulpo puede manejar múltiples ramas, pero es mucho más cuidadosa para evitar incurrir en complejos conflictos; y es por eso que se utiliza en los intentos de fusionar más de dos ramas.

Pero existen también otras estrategias que se pueden escoger según se necesiten. Una de ellas, la *fusión subárbol* (*subtree merge*), es precisamente la más adecuada para tratar con subproyectos. En este caso se va a mostrar cómo se haría el mismo empotramiento del módulo rack tomado como ejemplo anteriormente, pero utilizando fusiones de subárbol en lugar de submódulos.

La idea subyacente tras toda fusión subarborea es la de que se tienen dos proyectos; y uno de ellos está relacionado con una subcarpeta en el otro, y viceversa. Cuando se solicita una fusión subarborea, Git es lo suficientemente inteligente como para imaginarse por sí solo que uno de los proyectos es un subárbol del otro y obrar en consecuencia. Es realmente sorprendente.



Se comienza añadiendo la aplicación Rack al proyecto. Se añade como una referencia remota en el propio proyecto, y luego se extrae (checkout) en su propia rama:

```
$ git remote add rack_remote git@github.com:schacon/rack.git
$ git fetch rack_remote
warning: no common commits
remote: Counting objects: 3184, done.
remote: Compressing objects: 100% (1465/1465), 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.Resolving deltas: 100% (1952/1952),
done.Resolving deltas: 100% (1952/1952), done.
From git@github.com:schacon/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"
```

En este punto, se tiene la raíz del proyecto Rack en la rama **rack\_branch** y la del propio proyecto padre en la rama **master**. Si se comprueban una o la otra, se puede observar que ambos proyectos tienen distintas raíces:

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

Si se desea situar el proyecto Rack como una subcarpeta del proyecto **master**. Se ha de lanzar el comando **git read-tree**. Se verá más en detalle el comando **read-tree** y sus acompañantes en el capítulo 9. Pero por ahora, basta con saber que este comando se encarga de leer el árbol raíz de una rama en el área de preparación (staging area) y carpeta de trabajo (working directory)

actuales. Con ello, se retorna sobre la rama **master** y se recupera (pull) la rama **rack\_branch** en la subcarpeta **rack** de la rama **master** del proyecto principal:

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

Cuando se confirman estos cambios, es como si se tuvieran todos los archivos Rack bajo esa carpeta --como si se hubieran copiado desde un archivo comprimido tarball-- Lo que hace interesante este método es la posibilidad que brinda de fusionar cambios de una rama sobre la otra de forma sencilla. De tal forma que, si se actualiza el proyecto Rack, se pueden integrar los cambios aguas arriba simplemente cambiando a esa rama y recuperando:

```
$ git checkout rack_branch
$ git pull
```

Tras lo cual, es posible fusionar esos cambios de vuelta a la rama 'master'. Utilizando el comando **git merge -s subtree**, que funciona correctamente; pero fusionando también los historiales entre sí. Un efecto secundario que posiblemente no interese. Para recuperar los cambios y rellenar el mensaje de la confirmación, se pueden emplear las opciones **--squash** y **--no-commit**, junto con la opción de estrategia **-s subtree**:

```
$ git checkout master
$ git merge --squash -s subtree --no-commit rack_branch
Squash commit -- not updating HEAD
Automatic merge went well; stopped before committing as requested
```

Con esto, todos los cambios en el proyecto Rack se encontrarán fusionados y listos para ser confirmados localmente. También es posible hacer el camino contrario: realizar los cambios en la subcarpeta **rack** de la rama 'master', para posteriormente fusionarlos en la rama **rack\_branch** y remitirlos a los encargados del mantenimiento o enviarlos aguas arriba.

Para ver las diferencias entre el contenido de la subcarpeta **rack** y el código en la rama **rack\_branch** --para comprobar si es necesario fusionarlas--, no se puede emplear el comando **diff** habitual. En su lugar, se ha de emplear el comando **git diff-tree** con la rama que se desea comparar:

```
$ git diff-tree -p rack_branch
```

O, otro ejemplo: para comparar el contenido de la subcarpeta **rack** con la rama **master** en el servidor:

```
$ git diff-tree -p rack_remote/master
```

# Chapter 7

## Personalizando Git

Hasta ahora, hemos visto los aspectos básicos del funcionamiento de Git y la manera de utilizarlo; además de haber presentado una serie de herramientas suministradas con Git para ayudarnos a usarlo de manera sencilla y eficiente. En este capítulo, avanzaremos sobre ciertas operaciones que puedes utilizar para personalizar el funcionamiento de Git ; presentando algunos de sus principales ajustes y el sistema de anclajes (hooks). Con estas operaciones, será fácil conseguir que Git trabaje exactamente como tú, tu empresa o tu grupo necesiteis.

### 7.1 Personalizando Git - Configuración de Git

#### Configuración de Git

Como se ha visto brevemente en el capítulo 1, podemos acceder a los ajustes de configuración de Git a través del comando 'git config'. Una de las primeras acciones que has realizado con Git ha sido el configurar tu nombre y tu dirección de correo-e.

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

Ahora vas a aprender un puñado de nuevas e interesantes opciones que puedes utilizar para personalizar el uso de Git.

Primeramente, vamos a repasar brevemente los detalles de configuración de Git que ya has visto en el primer capítulo. Para determinar su comportamiento no estandar, Git emplea una serie de archivos de configuración. El primero de ellos es el archivo '/etc/gitconfig', que contiene valores para todos y cada uno de los usuarios en el sistema y para todos sus repositorios. Con la opción '--system' del comando 'git config', puedes leer y escribir de/a este archivo.

El segundo es el archivo '~/.gitconfig', específico para cada usuario. Con la opción '--global', 'git config' lee y escribe en este archivo.

Y por último, Git también puede considerar valores de configuración presentes en el archivo '.git/config' de cada repositorio que estés utilizando. Estos valores se aplicarán únicamente a dicho repositorio. Cada nivel sobrescribe los valores del nivel anterior; es decir lo configurado en '.git/config' tiene primacia con respecto a lo configurado en '/etc/gitconfig', por ejemplo. También puedes ajustar estas configuraciones manualmente, editando directamente los archivos correspondientes y escribiendo en ellos con la sintaxis correspondiente; pero suele ser más sencillo hacerlo siempre a través del comando 'git config'.

## **Configuración básica del cliente Git**

Las opciones de configuración reconocidas por Git pueden distribuirse en dos grandes categorías: las del lado cliente y las del lado servidor. La mayoría de las opciones están en el lado cliente, --configurando tus preferencias personales de trabajo--. Aunque hay multitud de ellas, aquí vamos a ver solamente unas pocas. Las mas comunmente utilizadas o las que afectan significativamente a tu forma de trabajar. No vamos a revisar aquellas opciones utilizadas solo en casos muy especiales. Si quieres consultar una lista completa, con todas las opciones contempladas en tu versión de Git, puedes lanzar el comando:

```
$ git config --help
```

La página de manual sobre 'git config' contiene una lista bastante detallada de todas las opciones disponibles.

### **core.editor**

Por defecto, Git utiliza cualquier editor que hayas configurado como editor de texto por defecto de tu sistema. O, si no lo has configurado, utilizará Vi como editor para crear y editar las etiquetas y mensajes de tus confirmaciones de cambio (commit). Para cambiar ese comportamiento, puedes utilizar el ajuste 'core.editor':

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

A partir de ese comando, por ejemplo, git lanzará Emacs cada vez que vaya a editar mensajes; indistintamente del editor configurado en la línea de comandos (shell) del sistema.

### **commit.template**

Si preparas este ajuste para apuntar a un archivo concreto de tu sistema, Git lo utilizará como mensaje por defecto cuando hagas confirmaciones de cambio. Por ejemplo, imagina que creas una plantilla en '\$HOME/.gitmessage.txt'; con un contenido tal como:

subject line

what happened

[ticket: X]

Para indicar a Git que lo utilice como mensaje por defecto y que aparezca en tu editor cuando lances el comando 'git commit', tan solo has de ajustar 'commit.template':

```
$ git config --global commit.template $HOME/.gitmessage.txt
$ git commit
```

A partir de entonces, cada vez que confirmes cambios (commit), tu editor se abrirá con algo como esto:

subject line

what happened

[ticket: X]

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   modified:   lib/test.rb
#
~
~
".git/COMMIT_EDITMSG" 14L, 297C
```

Si tienes una política concreta con respecto a los mensajes de confirmación de cambios, puedes aumentar las posibilidades de que sea respetada si creas una plantilla acorde a dicha política y la pones como plantilla por defecto de Git.

### **core.pager**

El parámetro `core.pager` selecciona el paginador utilizado por Git cuando muestra resultados de comandos tales como `'log'` o `'diff'`. Puedes ajustarlo para que utilice `'more'` o tu paginador favorito, (por defecto, se utiliza `'less'`); o puedes anular la paginación si le asignas una cadena vacía.

```
$ git config --global core.pager "
```

Si lanzas esto, Git mostrará siempre el resultado completo de todos los comandos, independientemente de lo largo que sea este.

### **user.signingkey**

Si tienes costumbre de firmar tus etiquetas (tal y como se ha visto en el capítulo 2), configurar tu clave de firma GPG puede facilitarte la labor. Configurando tu clave ID de esta forma:

```
$ git config --global user.signingkey <gpg-key-id>
```

Puedes firmar etiquetas sin necesidad de indicar tu clave cada vez en el comando `'git tag'`.

```
$ git tag -s <tag-name>
```

### **core.excludesfile**

Se pueden indicar expresiones en el archivo `'.gitignore'` de tu proyecto para indicar a Git lo que debe considerar o no como archivos sin seguimiento, o lo que interará o no seleccionar cuando lances el comando `'git add'`, tal y como se indicó en el capítulo 2. Sin embargo, si quieres disponer de otro archivo fuera de tus proyectos o tener expresiones extra, puedes indicárselo a Git con el parámetro `'core.excludesfile'`. Simplemente, configúralo para que apunte a un archivo con contenido similar al que tendría cualquier archivo `'.gitignore'`.

### **help.autocorrect**

Este parámetro solo está disponible a partir de la versión 1.6.1 de Git. Cada vez que tienes un error de tecleo en un comando, Git 1.6 te muestra algo como:

```
$ git com
git: 'com' is not a git-command. See 'git --help'.
```

```
Did you mean this?
    commit
```

Si ajustas 'help.autocorrect' a 1, Git lanzará automáticamente el comando corregido, (pero solo cuando haya únicamente uno que pueda encajar).

## Colores en Git

Git puede marcar con colores los resultados que muestra en tu terminal, ayudandote así a leerlos más fácilmente. Hay unos cuantos parámetros que te pueden ayudar a configurar tus colores favoritos.

### **color.ui**

Si se lo pides, Git coloreará automáticamente la mayor parte de los resultados que muestre. Puedes ajustar con precisión cada una de las partes a colorear; pero si deseas activar de un golpe todos los colores por defecto, no tienes más que poner a "true" el parámetro 'color.ui'.

```
$ git config --global color.ui true
```

Ajustando así este parámetro, Git colorea sus resultados cuando estos se muestran en un terminal. Otros ajustes posibles son "false", para indicar a Git no colorear nunca ninguno de sus resultados; y "always", para indicar colorear siempre, incluso cuando se redirija la salida a un archivo o a otro comando. Este parámetro se añadió en la versión 1.5.5 de Git. Si tienes una versión más antigua, tendrás que indicar específicamente todos y cada uno de los colores individualmente.

Será muy raro ajustar 'color.ui = always'. En la mayor parte de las ocasiones, cuando necesites códigos de color en los resultados, es mejor indicar puntualmente la opción '--color' en el comando concreto, para obligarle a utilizar códigos de color. Habitualmente, se trabajará con el ajuste 'color.ui = true'.

### **color.\***

Cuando quieras ajustar específicamente, comando a comando, donde colorear y cómo colorear, (o cuando tengas una versión antigua de Git), puedes emplear los ajustes particulares de color. Cada uno de ellos puede fijarse a 'true' (verdadero), 'false' (falso) o 'always' (siempre):

color.branch

color.diff

color.interactive

color.status



Además, cada uno de ellos tiene parámetros adicionales para asignar colores a partes específicas, por si quieres precisar aún más. Por ejemplo, para mostrar la meta-información del comando 'diff' con letra azul sobre fondo negro y con caracteres en negrita, puedes indicar:

```
$ git config --global color.diff.meta "blue black bold"
```

Puedes ajustar un color a cualquiera de los siguientes valores: 'normal' (normal), 'black' (negro), 'green' (verde), 'yellow' (amarillo), 'blue' (azul oscuro), 'magenta' (rojo oscuro), 'cyan' (azul claro) o 'white' (blanco). También puedes aplicar atributos tales como 'bold' (negrita), 'dim' (tenue), 'ul' ( ), 'blink' (parpadeante) y 'reverse' (video inverso).

Mira en la página man de 'git config' si deseas tener explicaciones más detalladas.

## Herramientas externas para fusionar y para comparar

Aunque Git lleva una implementación interna de diff, la que se utiliza habitualmente, se puede sustituir por una herramienta externa. Puedes incluso configurar una herramienta gráfica para la resolución de conflictos, en lugar de resolverlos manualmente. Lo voy a demostrar configurando Perforce Visual Merge como herramienta para realizar las comparaciones y resolver conflictos; ya que es una buena herramienta gráfica y es libre.

Si lo quieres probar, P4Merge funciona en todas las principales plataformas. Los nombres de carpetas que utilizaré en los ejemplos funcionan en sistemas Mac y Linux; para Windows, tendrás que sustituir '/usr/local/bin' por el correspondiente camino al ejecutable en tu sistema.

P4Merge se puede descargar desde:

<http://www.perforce.com/product/components/perforce-visual-merge-and-diff-tools>

Para empezar, tienes que preparar los correspondientes scripts para lanzar tus comandos. En estos ejemplos, voy a utilizar caminos y nombres Mac para los ejecutables; en otros sistemas, tendrás que sustituirlos por los correspondientes donde tengas instalado 'p4merge'. El primer script a preparar es uno al que denominaremos 'extMerge', para llamar al ejecutable con los correspondientes argumentos:

```
$ cat /usr/local/bin/extMerge
#!/bin/sh
/Applications/p4merge.app/Contents/MacOS/p4merge $*
```

El script para el comparador, ha de asegurarse de recibir siete argumentos y de pasar dos de ellos al script de fusion (merge). Por defecto, Git pasa los siguientes argumentos al programa diff (comparador):

```
path old-file old-hex old-mode new-file new-hex new-mode
```

Ya que solo necesitarás 'old-file' y 'new-file', puedes utilizar el siguiente script para extraerlos:

```
$ cat /usr/local/bin/extDiff
#!/bin/sh
[ $# -eq 7 ] && /usr/local/bin/extMerge "$2" "$5"
```

Además has de asegurarte de que estas herramientas son ejecutables:

```
$ sudo chmod +x /usr/local/bin/extMerge
$ sudo chmod +x /usr/local/bin/extDiff
```

Una vez preparado todo esto, puedes ajustar el archivo de configuración para utilizar tus herramientas personalizadas de comparación y resolución de conflictos. Tenemos varios parámetros a ajustar: 'merge.tool' para indicar a Git la estrategia que ha de usar, **mergetool.\*.cmd** para especificar como lanzar el comando, 'mergetool.trustExitCode' para decir a Git si el código de salida del programa indica una fusión con éxito o no, y 'diff.external' para decir a Git qué comando lanzar para realizar comparaciones. Es decir, has de ejecutar cuatro comandos de configuración:

```
$ git config --global merge.tool extMerge
$ git config --global mergetool.extMerge.cmd \
    'extMerge "$BASE" "$LOCAL" "$REMOTE" "$MERGED"'
$ git config --global mergetool.trustExitCode false
$ git config --global diff.external extDiff
```

o puedes editar tu archivo '~/.gitconfig' para añadirle las siguientes lineas:

```
[merge]
  tool = extMerge
[mergetool "extMerge"]
  cmd = extMerge \"$BASE\" \"$LOCAL\" \"$REMOTE\" \"$MERGED\"
  trustExitCode = false
[diff]
```

external = extDiff

Tras ajustar todo esto, si lanzas comandos tales como: `$ git diff 32d1776b1 ^ 32d1776b1`

En lugar de mostrar las diferencias por línea de comandos, Git lanzará P4Merge, que tiene una pinta como la de la Figura 7-1.

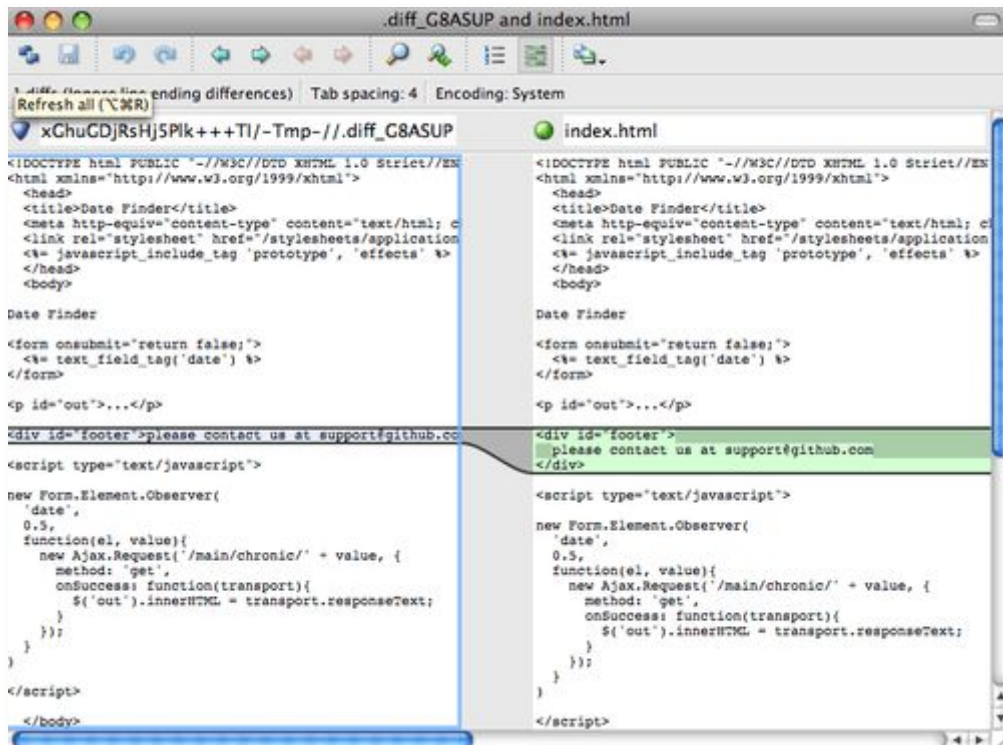


Figura 7-1. P4Merge.

Si intentas fusionar (merge) dos ramas y tienes los consabidos conflictos de integración, puedes lanzar el comando `'git mergetool'`; lanzará P4Merge para ayudarte a resolver los conflictos por medio de su interfaz gráfica.

Lo bonito de estos ajustes con scripts, es que puedes cambiar facilmente tus herramientas de comparación (diff) y de fusión (merge). Por ejemplo, para cambiar tus scripts `'extDiff'` y `'extMerge'` para utilizar KDiff3, tan solo has de editar el archivo `'extMerge'`:

```
$ cat /usr/local/bin/extMerge
#!/bin/sh
/Applications/kdiff3.app/Contents/MacOS/kdiff3 $*
```

A partir de ahora, Git utilizará la herramienta KDiff3 para mostrar y resolver conflictos de integración.

Git viene preparado para utilizar bastantes otras herramientas de resolución de conflictos, sin necesidad de andar ajustando la configuración de cdm. Puedes utilizar kdiff3, opendiff, tkdiff, meld, xxdiff, emerge, vimdiff, o gvimdiff como herramientas de fusión. Por ejemplo, si no te interesa utilizar KDiff3 para comparaciones, sino que tan solo te interesa utilizarlo para resolver conflictos de integración; teniendo kdiff3 en el path de ejecución, solo has de lanzar el comando:

```
$ git config --global merge.tool kdiff3
```

Si utilizas este comando en lugar de preparar los archivos 'extMerge' y 'extDiff' antes comentados, Git utilizará KDiff3 para resolución de conflictos de integración y la herramienta estándar diff para las comparaciones.

### Formato y espacios en blanco

El formato y los espacios en blanco son la fuente de los problemas más sutiles y frustrantes que muchos desarrolladores se pueden encontrar en entornos colaborativos, especialmente si son multi-plataforma. Es muy fácil que algunos parches u otro trabajo recibido introduzcan sutiles cambios de espaciado, porque los editores suelen hacerlo inadvertidamente o, trabajando en entornos multi-plataforma, porque programadores Windows suelen añadir retornos de carro al final de las líneas que tocan. Git dispone de algunas opciones de configuración para ayudarnos con estos problemas.

#### **core.autocrlf**

Si estás programando en Windows o utilizando algún otro sistema, pero colaborando con gente que programa en Windows. Es muy posible que alguna vez te topes con problemas de finales de línea. Esto se debe a que Windows utiliza retorno-de-carro y salto-de-línea para marcar los finales de línea de sus archivos. Mientras que Mac y Linux utilizan solamente el carácter de salto-de-línea. Esta es una sutil, pero molesta, diferencia cuando se trabaja en entornos multi-plataforma.

Git la maneja autoconvirtiendo los finales CRLF en LF al hacer confirmaciones de cambios (commit); y, viceversa, al extraer código (checkout) a la carpeta de trabajo. Puedes activar esta funcionalidad con el parámetro 'core.autocrlf'. Si estás trabajando en una máquina Windows, ajústalo a 'true', --para convertir finales LF en CRLF cuando extraigas código (checkout)--.

```
$ git config --global core.autocrlf true
```

Si estás trabajando en una máquina Linux o Mac, entonces no te interesa convertir automáticamente los finales de línea al extraer código. Sino que te interesa arreglar los posibles CRLF que pudieran aparecer accidentalmente. Puedes indicar a Git que convierta CRLF en LF al confirmar cambios (commit), pero no en el otro sentido; utilizando también el parámetro 'core.autocrlf':

```
$ git config --global core.autocrlf input
```

Este ajuste dejará los finales de línea CRLF en las extracciones de código (checkout), pero los finales LF en sistemas Mac o Linux y en el repositorio.

Si eres un programador Windows, trabajando en un entorno donde solo haya máquinas Windows, puedes desconectar esta funcionalidad. Para almacenar CRLFs en el repositorio. Ajustando el parámetro a 'false':

```
$ git config --global core.autocrlf false
```

### **core.whitespace**

Git viene preajustado para detectar y resolver algunos de los problemas más típicos relacionados con los espacios en blanco. Puede vigilar acerca de cuatro tipos de problemas de espaciado --dos los tiene activados por defecto, pero se pueden desactivar; y dos vienen desactivados por defecto, pero se pueden activar--.

Los dos activos por defecto son 'trailing-space' (espaciado de relleno), que vigila por si hay espacios al final de las líneas, y 'space-before-tab' (espaciado delante de un tabulador), que mira por si hay espacios al principio de las líneas o por delante de los tabuladores.

Los dos inactivos por defecto son 'indent-with-non-tab' (indentado sin tabuladores), que vigila por si alguna línea empieza con ocho o mas espacios en lugar de con tabuladores; y 'cr-at-eol' (retorno de carro al final de línea), que vigila para que haya retornos de carro en todas las líneas.

Puedes decir a Git cuales de ellos deseas activar o desactivar, ajustando el parámetro 'core.whitespace' con los valores on/off separados por comas. Puedes desactivarlos tanto dejándolos fuera de la cadena de ajustes, como añadiendo el prefijo '-' delante del valor. Por ejemplo, si deseas activar todos menos 'cr-at-eol' puedes lanzar:

```
$ git config --global core.whitespace \
    trailing-space,space-before-tab,indent-with-non-tab
```

Git detectará posibles problemas cuando lance un comando 'git diff', e intentará destacarlos en otro color para que puedas corregirlos antes de confirmar cambios (commit). También pueden ser útiles estos ajustes cuando estás incorporando parches con 'git apply'. Al incorporar parches, puedes pedirle a Git que te avise específicamente sobre determinados problemas de espaciado:

```
$ git apply --whitespace=warn <patch>
```

O puedes pedirle que intente corregir automáticamente los problemas antes de aplicar el parche:

```
$ git apply --whitespace=fix <patch>
```

Estas opciones se pueden aplicar también al comando 'git rebase'. Si has confirmado cambios con problemas de espaciado, pero no los has enviado (push) aún "aguas arriba". Puedes realizar una reorganización (rebase) con la opción '--whitespace=fix' para que Git corrija automáticamente los problemas según va reescribiendo los parches.

## Configuración de Servidor

No hay tantas opciones de configuración en el lado servidor de Git. Pero hay unas pocas interesantes que merecen ser tenidas en cuenta.

### **receive.fsckObjects**

Por defecto, Git no suele comprobar la consistencia de todos los objetos que recibe durante un envío (push). Aunque Git tiene la capacidad para asegurarse de que cada objeto sigue casando con su suma de control SHA-1 y sigue apuntando a objetos válidos. No lo suele hacer en todos y cada uno de los envíos (push). Es una operación costosa, que, dependiendo del tamaño del repositorio, puede llegar a añadir mucho tiempo a cada operación de envío (push). De todas formas, si deseas que Git compruebe la consistencia de todos los objetos en todos los envíos, puedes forzarle a hacerlo ajustando a 'true' el parámetro 'receive.fsckObjects':

```
$ git config --system receive.fsckObjects true
```

A partir de ese momento, Git comprobará la integridad del repositorio antes de aceptar ningún envío (push), para asegurarse de que no está introduciendo datos corruptos.

### **receive.denyNonFastForwards**

Si reorganizas (rebase) confirmaciones de cambio (commit) que ya habías enviado y tratas de enviarlas (push) de nuevo. O si intentas enviar una confirmación a una rama remota que no contiene la confirmación actualmente apuntada por la rama. Normalmente, la operación te será denegada por la rama remota sobre la que pretendías realizarla. Habitualmente, este es el comportamiento más adecuado. Pero, en el caso de las reorganizaciones, cuando estás totalmente seguro de lo que haces, puedes forzar el envío, utilizando la opción '-f' en el comando 'git push' a la rama remota.

Para impedir estos envíos forzados de referencias de avance no directo (no fast-forward) a ramas remotas, es para lo que se emplea el parámetro 'receive.denyNonFastForwards':

```
$ git config --system receive.denyNonFastForwards true
```

Otra manera de obtener el mismo resultado, es a través de los enganches (hooks) en el lado servidor. Enganches de los que hablaremos en breve. Esta otra vía te permite realizar ajustes más finos, tales como denegar referencias de avance no directo, (non-fast-forwards), únicamente a un grupo de usuarios.

### **receive.denyDeletes**

Uno de los cortocircuitos que suelen utilizar los usuarios para saltarse la política de 'denyNonFastForwards', suele ser el borrar la rama y luego volver a enviarla de vuelta con la nueva referencia. En las últimas versiones de Git (a partir de la 1.6.1), se puede evitar poniendo a 'true' el parámetro 'receive.denyDeletes':

```
$ git config --system receive.denyDeletes true
```

Esto impide el borrado de ramas o de etiquetas por medio de un envío a través de la mesa (push across the board), --ningún usuario lo podrá hacer--. Para borrar ramas remotas, tendrás que borrar los archivos de referencia manualmente sobre el propio servidor. Existen también algunas otras maneras más interesantes de hacer esto mismo, pero para usuarios concretos, a través de permisos (ACLs); tal y como veremos al final de este capítulo.

## 7.2 Personalizando Git - Atributos de Git

## Atributos de Git

Algunos de los ajustes que hemos vistos, pueden ser especificados para un camino (path) concreto, de tal forma que Git los aplicará únicamente para una carpeta o para un grupo de archivos determinado. Estos ajustes específicos relacionados con un camino, se denominan atributos en Git. Y se pueden fijar, bien mediante un archivo `'.gitattribute'` en uno de los directorios de tu proyecto (normalmente en la raíz del proyecto), o bien mediante el archivo `'git/info/attributes'` en el caso de no querer guardar el archivo de atributos dentro de tu proyecto.

Por medio de los atributos, puedes hacer cosas tales como indicar diferentes estrategias de fusión para archivos o carpetas concretas de tu proyecto, decirle a Git cómo comparar archivos no textuales, o indicar a Git que filtre ciertos contenidos antes de guardarlos o de extraerlos del repositorio Git. En esta sección, aprenderás acerca de algunos atributos que puedes asignar a ciertos caminos (paths) dentro de tu proyecto Git, viendo algunos ejemplos de cómo utilizar sus funcionalidades de manera práctica.

### Archivos binarios

Un buen truco donde utilizar los atributos Git es para indicarle cuales de los archivos son binarios, (en los casos en que Git no podría llegar a determinarlo por sí mismo), dándole a Git instrucciones especiales sobre cómo tratar estos archivos. Por ejemplo, algunos archivos de texto se generan automáticamente y no tiene sentido compararlos; mientras que algunos archivos binarios sí que pueden ser comparados --vamos a ver cómo indicar a Git cual es cual--.

#### **Identificando archivos binarios**

Algunos archivos aparentan ser textuales, pero a efectos prácticos merece más la pena tratarlos como binarios. Por ejemplo, los proyectos Xcode en un Mac contienen un archivo terminado en `'.pbxproj'`. Este archivo es básicamente una base de datos JSON (datos javascript en formato de texto plano), escrita directamente por el IDE para almacenar aspectos tales como tus ajustes de compilación. Aunque técnicamente es un archivo de texto, porque su contenido son caracteres ASCII. Realmente nunca lo tratarás como tal, porque en realidad es una base de datos ligera --y no puedes fusionar sus contenidos si dos personas lo cambian, porque las comparaciones no son de utilidad--. Estos son archivos destinados a ser tratados de forma automatizada. Y es preferible tratarlos como si fueran archivos binarios.

Para indicar a Git que trate todos los archivos `'pbxproj'` como binarios, puedes añadir esta línea a tu archivo `'.gitattributes'`:



**\*.pbxproj -crlf -diff**

A partir de ahora, Git no intentará convertir ni corregir problemas CRLF en los finales de línea; ni intentará hacer comparaciones ni mostrar diferencias de este archivo cuando lances comandos 'git show' o 'git diff' en tu proyecto. A partir de la versión 1.6 de Git, puedes utilizar una macro en lugar de las dos opciones '-crlf -diff':

**\*.pbxproj binary**

### **Comparando archivos binarios**

A partir de la versión 1.6, puedes utilizar los atributos Git para comparar archivos binarios. Se consigue diciéndole a Git la forma de convertir los datos binarios en texto, consiguiendo así que puedan ser comparado con la herramienta habitual de comparación textual.

Esta es una funcionalidad muy útil, pero bastante desconocida. Por lo que la ilustraré con unos ejemplos. En el primero de ellos, utilizarás esta técnica para resolver uno de los problemas más engorrosos conocidos por la humanidad: el control de versiones en documentos Word. Todo el mundo conoce el hecho de que Word es el editor más horroroso de cuantos hay; pero, desgraciadamente, todo el mundo lo usa. Si deseas controlar versiones en documentos Word, puedes añadirlos a un repositorio Git e ir realizando confirmaciones de cambio (commit) cada vez. Pero, ¿qué ganas con ello?. Si lanzas un comando 'git diff', lo único que verás será algo tal como:

```
$ git diff
diff --git a/chapter1.doc b/chapter1.doc
index 88839c4..4afcb7c 100644
Binary files a/chapter1.doc and b/chapter1.doc differ
```

No puedes comparar directamente dos versiones, a no ser que extraigas ambas y las compares manualmente, ¿no?. Pero resulta que puedes hacerlo bastante mejor utilizando los atributos Git. Poniendo lo siguiente en tu archivo '.gitattributes':

**\*.doc diff=word**

Así decimos a Git que sobre cualquier archivo coincidente con el patrón indicado, (.doc), ha de utilizar el filtro "word" cuando intentente hacer una comparación con él. ¿Qué es el filtro "word"? Tienes que configurarlo tú mismo. Por ejemplo, puedes configurar Git para que utilice el

programa 'strings' para convertir los documentos Word en archivos de texto planos, archivos sobre los que poder realizar comparaciones sin problemas:

```
$ git config diff.word.textconv strings
```

A partir de ahora, Git sabe que si intenta realizar una comparación entre dos momentos determinados (snapshots), y si cualquiera de los archivos a comparar termina en '.doc', tiene que pasar antes esos archivos por el filtro "word", es decir, por el programa 'strings'. Esto prepara versiones texto de los archivos Word, antes de intentar compararlos.

Un ejemplo. He puesto el capítulo 1 de este libro en Git, le he añadido algo de texto a un párrafo y he guardado el documento. Tras lo cual he lanzando el comando 'git diff' para ver lo que ha cambiado:

```
$ git diff
diff --git a/chapter1.doc b/chapter1.doc
index c1c8a0a..b93c9e4 100644
--- a/chapter1.doc
+++ b/chapter1.doc
@@ -8,7 +8,8 @@ re going to cover Version Control Systems (VCS) and Git basics
 re going to cover how to get it and set it up for the first time if you don
 t already have it on your system.
 In Chapter Two we will go over basic Git usage - how to use Git for the 80%
-s going on, modify stuff and contribute changes. If the book spontaneously
+s going on, modify stuff and contribute changes. If the book spontaneously
+Let's see if this works.
```

Git me indica correctamente que he añadido la frase "Let's see if this works". No es perfecto, --añade bastante basura aleatoria al final--, pero realmente funciona. Si pudieras encontrar o escribir un conversor suficientemente bueno de-Word-a-texto-plano, esta solución sería terriblemente efectiva. Sin embargo, ya que 'strings' está disponible para la mayor parte de los sistemas Mac y Linux, es buena idea probar primero con él para trabajar con formatos binarios.

Otro problema donde puede ser útil esta técnica, es en la comparación de imágenes. Un camino puede ser pasar los archivos JPEG a través de un filtro para extraer su información EXIF --los metadatos que se graban dentro de la mayoría de formatos gráficos--. Si te descargas e instalas el programa 'exiftool', puedes utilizarlo para convertir tus imagenes a textos (metadatos), de tal forma que diff podrá al menos mostrarte algo útil de cualquier cambio que se produzca:

```
$ echo '*.png diff=exif' >> .gitattributes
$ git config diff.exif.textconv exiftool
```

Si sustituyes alguna de las imagenes en tu proyecto, y lanzas el comando 'git diff' obtendrás algo como:

```
diff --git a/image.png b/image.png
index 88839c4..4afcb7c 100644
--- a/image.png
+++ b/image.png
@@ -1,12 +1,12 @@
ExifTool Version Number      : 7.74
-File Size                   : 70 kB
-File Modification Date/Time : 2009:04:21 07:02:45-07:00
+File Size                   : 94 kB
+File Modification Date/Time : 2009:04:21 07:02:43-07:00
File Type                    : PNG
MIME Type                    : image/png
-Image Width                 : 1058
-Image Height                : 889
+Image Width                 : 1056
+Image Height                : 827
Bit Depth                    : 8
Color Type                   : RGB with Alpha
```

Aquí se vé claramente que ha cambiado el tamaño del archivo y las dimensiones de la imagen.

## Expansión de palabras clave

Algunos usuarios de sistemas SVN o CVS, hechan de menos el disponer de expansiones de palabras clave al estilo de las que dichos sistemas tienen. El principal problema para hacerlo en Git reside en la imposibilidad de modificar los ficheros con información relativa a la confirmación de cambios (commit). Debido a que Git calcula sus sumas de comprobación antes de las confirmaciones. De todas formas, es posible inyectar textos en un archivo cuando lo extraemos del repositorio (checkout) y quitarlos de nuevo antes de devolverlo al repositorio (commit). Los atributos Git admiten dos maneras de realizarlo.

La primera, es inyectando automáticamente la suma de comprobación SHA-1 de un gran objeto binario (blob) en un campo '\$Id\$' dentro del archivo. Si colocas este atributo en un archivo o conjunto de archivos, Git lo sustituirá por la suma de comprobación SHA-1 la próxima vez que

lo/s extraiga/s. Es importante destacar que no se trata de la suma SHA de la confirmación de cambios (commit), sino del propio objeto binario (blob):

```
$ echo '*.txt ident' >> .gitattributes  
$ echo '$Id$' > test.txt
```

La próxima vez que extraigas el archivo, Git le habrá inyectado el SHA del objeto binario (blob):

```
$ rm test.txt  
$ git checkout -- test.txt  
$ cat test.txt  
$Id: 42812b7653c7b88933f8a9d6cad0ca16714b9bb3 $
```

Pero esto tiene un uso bastante limitado. Si has utilizado alguna vez las sustituciones de CVS o de Subversion, sabrás que pueden incluir una marca de fecha, --la suma de comprobación SHA no es igual de útil, ya que, por ser bastante aleatoria, es imposible deducir si una suma SHA es anterior o posterior a otra--.

Aunque resulta que también puedes escribir tus propios filtros para realizar sustituciones en los archivos al guardar o recuperar (commit/checkout). Esos son los filtros "clean" y "smudge". En el archivo '.gitattributes', puedes indicar filtros para carpetas o archivos determinados y luego preparar tus propios scripts para procesarlos justo antes de confirmar cambios en ellos ("clean", ver Figura 7-2), o justo antes de recuperarlos ("smudge", ver Figura 7-3). Estos filtros pueden utilizarse para realizar todo tipo de acciones útiles.

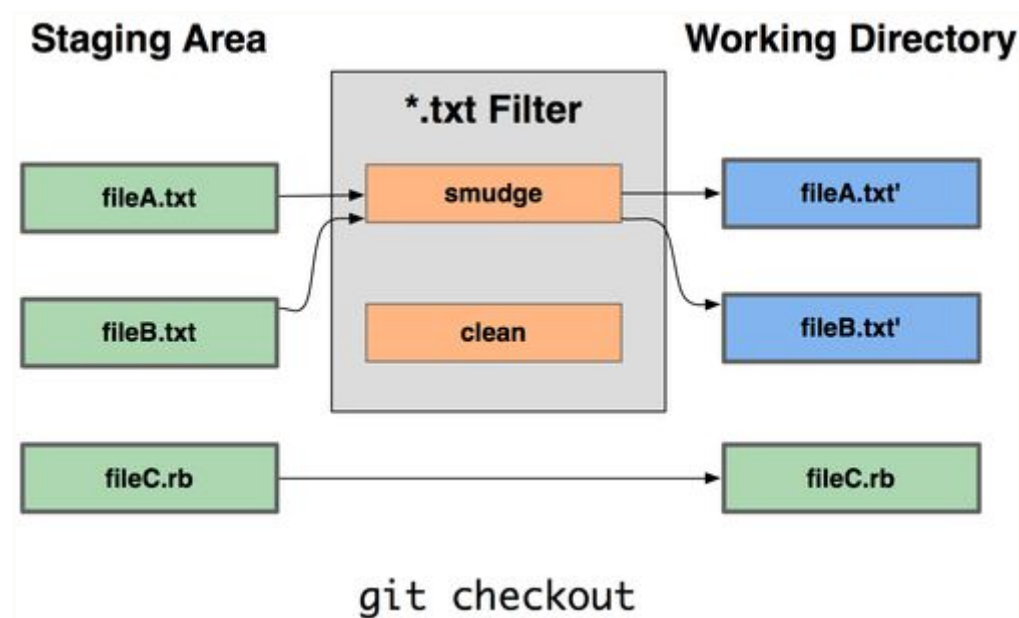


Figura 7-2. El filtro "smudge" se usa al extraer (checkout).

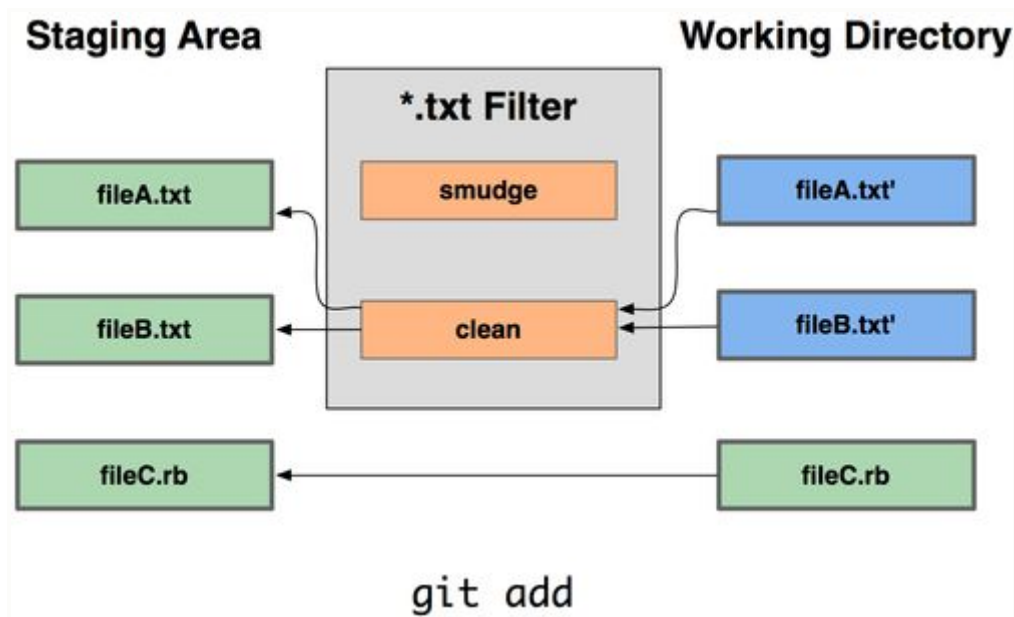


Figura 7-3. El filtro "clean" se usa al almacenar (staged).

El mensaje de confirmación para esta funcionalidad nos da un ejemplo simple: el de pasar todo tu código fuente C por el programa 'indent' antes de almacenarlo. Puedes hacerlo poniendo los atributos adecuados en tu archivo '.gitattributes', para filtrar los archivos `*.c` a través de "indent":

```
*.c  filter=indent
```

E indicando después que el filtro "indent" actuará al manchar (smudge) y al limpiar (clean):

```
$ git config --global filter.indent.clean indent
$ git config --global filter.indent.smudge cat
```

En este ejemplo, cuando confirmes cambios (commit) en archivos con extensión `*.c`, Git los pasará previamente a través del programa 'indent' antes de confirmarlos, y los pasará a través del programa 'cat' antes de extraerlos de vuelta al disco. El programa 'cat' es básicamente transparente: de él salen los mismos datos que entran. El efecto final de esta combinación es el de filtrar todo el código fuente C a través de 'indent' antes de confirmar cambios en él.

Otro ejemplo interesante es el de poder conseguir una expansión de la clave '\$Date\$' del estilo de RCS. Para hacerlo, necesitas un pequeño script que coja el nombre de un archivo, localice la fecha de la última confirmación de cambios en el proyecto, e inserte dicha información en el archivo. Este podría ser un pequeño script Ruby para hacerlo:

```
#!/usr/bin/env ruby
data = STDIN.read
last_date = `git log --pretty=format:"%ad" -1`
puts data.gsub('$Date$', '$Date: ' + last_date.to_s + '$')
```

Simplemente, utiliza el comando 'git log' para obtener la fecha de la última confirmación de cambios, y sustituye con ella todas las cadenas '\$Date\$' que encuentre en el flujo de entrada stdin; imprimiendo luego los resultados. --Debería de ser sencillo de implementarlo en cualquier otro lenguaje que domines.-- Puedes llamar 'expanddate' a este archivo y ponerlo en el path de ejecución. Tras ello, has de poner un filtro en Git (podemos llamarle 'dater'), e indicarle que use el filtro 'expanddate' para manchar (smudge) los archivos al extraerlos (checkout). Puedes utilizar una expresión Perl para limpiarlos (clean) al almacenarlos (commit):

```
$ git config filter.dater.smudge expand_date
$ git config filter.dater.clean 'perl -pe "s/\\$Date[^\\$]*\\$/\\$Date\\$/'"
```

Esta expresión Perl extrae cualquier cosa que vea dentro de una cadena '\$Date\$', para devolverla a como era en un principio. Una vez preparado el filtro, puedes comprobar su funcionamiento preparando un archivo que contenga la clave '\$Date\$' e indicando a Git cual es el atributo para reconocer ese tipo de archivo:

```
$ echo '# $Date$' > date_test.txt
$ echo 'date*.txt filter=dater' >> .gitattributes
```

Al confirmar cambios (commit) y luego extraer (checkout) el archivo de vuelta, verás la clave sustituida:

```
$ git add date_test.txt .gitattributes
$ git commit -m "Testing date expansion in Git"
$ rm date_test.txt
$ git checkout date_test.txt
$ cat date_test.txt
```

# \$Date: Tue Apr 21 07:26:52 2009 -0700\$

Esta es una muestra de lo poderosa que puede resultar esta técnica para aplicaciones personalizadas. No obstante, debes de ser cuidadoso, ya que el archivo '.gitattributes' se almacena y se transmite junto con el proyecto; pero no así el propio filtro, (en este caso, 'dater'), sin el cual no puede funcionar. Cuando diseñes este tipo de filtros, han de estar pensados para que el proyecto continúe funcionando correctamente incluso cuando fallen.

## Exportación del repositorio

Los atributos de Git permiten realizar algunas cosas interesantes cuando exportas un archivo de tu proyecto.

### **export-ignore**

Puedes indicar a Git que ignore y no exporte ciertos archivos o carpetas cuando genera un archivo de almacenamiento. Cuando tienes alguna carpeta o archivo que no deseas incluir en tus registros, pero quieras tener controlado en tu proyecto, puedes marcarlos a través del atributo 'export-ignore'.

Por ejemplo, digamos que tienes algunos archivos de pruebas en la carpeta 'test/', y que no tiene sentido incluirlos en los archivos comprimidos (tarball) al exportar tu proyecto. Puedes añadir la siguiente línea al archivo de atributos de Git:

test/ export-ignore

A partir de ese momento, cada vez que lances el comando 'git archive' para crear un archivo comprimido de tu proyecto, esa carpeta no se incluirá en él.

### **export-subst**

Otra cosa que puedes realizar sobre tus archivos es algún tipo de sustitución simple de claves. Git te permite poner la cadena '\$Format:\$' en cualquier archivo, con cualquiera de las claves de formateo de '--pretty=format' que vimos en el capítulo 2. Por ejemplo, si deseas incluir un archivo llamado 'LAST COMMIT' en tu proyecto, y poner en él automáticamente la fecha de la última confirmación de cambios cada vez que lances el comando 'git archive':

```
$ echo 'Last commit date: $Format:%cd$' > LAST_COMMIT
$ echo "LAST_COMMIT export-subst" >> .gitattributes
$ git add LAST_COMMIT .gitattributes
```

```
$ git commit -am 'adding LAST_COMMIT file for archives'
```

Cuando lances la orden 'git archive', lo que la gente verá en ese archivo cuando lo abra será:

```
$ cat LAST_COMMIT
```

```
Last commit date: $Format:Tue Apr 21 08:38:48 2009 -0700$
```

## Estrategias de fusión

También puedes utilizar los atributos Git para indicar distintas estrategias de fusión para archivos específicos de tu proyecto. Una opción muy útil es la que nos permite indicar a Git que no intente fusionar ciertos archivos concretos cuando tengan conflictos, manteniendo en su lugar tus archivos sobre los de cualquier otro.

Puede ser interesante si una rama de tu proyecto es divergente o esta especializada, pero deseas seguir siendo capaz de fusionar cambios de vuelta desde ella, e ignorar ciertos archivos. Digamos que tienes un archivo de datos denominado database.xml, distinto en las dos ramas, y que deseas fusionar en la otra rama sin perturbarlo. Puedes ajustar un atributo tal como:

```
database.xml merge=ours
```

Al fusionar con otra rama, en lugar de tener conflictos de fusión con el archivo database.xml, obtendrás algo como:

```
$ git merge topic
```

```
Auto-merging database.xml
```

```
Merge made by recursive.
```

Y el archivo database.xml permanecerá inalterado en cualquier que fuera la versión que tú tenías originalmente.

## 7.3 Personalizando Git - Puntos de enganche Git



## Puntos de enganche Git

Al igual que en otros sistemas de control de versiones, Git también cuenta con mecanismos para lanzar scripts de usuario cuando suceden ciertas acciones importantes. Hay dos grupos de esos puntos de lanzamiento: los del lado cliente y los del lado servidor. Los puntos del lado cliente están relacionados con operaciones tales como la confirmación de cambios (commit) o la fusión (merge). Los del lado servidor están relacionados con operaciones tales como la recepción de contenidos enviados (push) a un servidor. Estos puntos de enganche pueden utilizarse para multitud de aplicaciones. Vamos a ver unas pocas de ellas.

### Instalando un punto de enganche

Los puntos de enganche se guardan en la subcarpeta 'hooks' de la carpeta Git. En la mayoría de proyectos, estará en '.git/hooks'. Por defecto, esta carpeta contiene unos cuantos scripts de ejemplo. Algunos de ellos son útiles por sí mismos; pero su misión principal es la de documentar las variables de entrada para cada script. Todos los ejemplos se han escrito como scripts de shell, con algo de código Perl embebido en ellos. Pero cualquier tipo de script ejecutable que tenga el nombre adecuado puede servir igual de bien --los puedes escribir en Ruby o en Python o en cualquier lenguaje de scripting con el que trabajes--. En las versiones de Git posteriores a la 1.6, esos ejemplos tendrán un nombre acabado en .sample; y tendrás que renombrarlos. Para las versiones anteriores a la 1.6, los ejemplos tienen el nombre correcto, pero les falta la marca de ejecutables.

Para activar un punto de enganche para un script, pon el archivo correspondiente en la carpeta 'hooks'; con el nombre adecuado y con la marca de ejecutable. A partir de ese momento, será automáticamente lanzado cuando se dé la acción correspondiente. Vamos a ver la mayoría de nombres de puntos de enganche disponibles.

### Puntos de enganche del lado cliente

Hay muchos de ellos. En esta sección los dividiremos en puntos de enganche en el flujo de trabajo de confirmación de cambios, puntos en el flujo de trabajo de correo electrónico y resto de puntos de enganche del lado servidor.

#### **Puntos en el flujo de trabajo de confirmación de cambios**

Los primeros cuatro puntos de enganche están relacionados con el proceso de confirmación de cambios. Primero se activa el punto de enganche 'pre-commit', incluso antes de que teclees el mensaje de confirmación. Se suele utilizar para inspeccionar la instantánea (snapshot) que vas a

confirmar, para ver si has olvidado algo, para asegurar que las pruebas se ejecutan, o para revisar cualquier aspecto que necesites inspeccionar en el código. Saliendo con un valor de retorno distinto de cero, se aborta la confirmación de cambios. Aunque siempre puedes saltartelo con la orden `'git commit --no-verify'`. Puede ser útil para realizar tareas tales como revisar el estilo del código (lanzando `'lint'` o algo equivalente), revisar los espacios en blanco de relleno (el script de ejemplo hace exactamente eso), o revisar si todos los nuevos métodos llevan la adecuada documentación.

El punto de enganche `'prepare-commit-msg'` se activa antes de arrancar el editor del mensaje de confirmación de cambios, pero después de crearse el mensaje por defecto. Te permite editar el mensaje por defecto, antes de que lo vea el autor de la confirmación de cambios. Este punto de enganche recibe varias entradas: la ubicación (path) del archivo temporal donde se almacena el mensaje de confirmación, el tipo de confirmación y la clave SHA-1 si estamos enmendando un commit existente. Este punto de enganche no tiene mucha utilidad para las confirmaciones de cambios normales; pero sí para las confirmaciones donde el mensaje por defecto es autogenerado, como en las confirmaciones de fusiones (merge), los mensajes con plantilla, las confirmaciones aplastadas (squash), o las confirmaciones de corrección (amend). Se puede utilizar combinándolo con una plantilla de confirmación, para poder insertar información automáticamente.

El punto de enganche `'commit-msg'` recibe un parámetro: la ubicación (path) del archivo temporal que contiene el mensaje de confirmación actual. Si este script termina con un código de salida distinto de cero, Git aborta el proceso de confirmación de cambios; permitiendo así validar el estado del proyecto o el mensaje de confirmación antes de permitir continuar. En la última parte de este capítulo, veremos cómo podemos utilizar este punto de enganche para revisar si el mensaje de confirmación es conforme a un determinado patrón obligatorio.

Después de completar todo el proceso de confirmación de cambios, es cuando se lanza el punto de enganche `'post-commit'`. Este no recibe ningún parámetro, pero podemos obtener fácilmente la última confirmación de cambios con el comando `'git log -1 HEAD'`. Habitualmente, este script final se suele utilizar para realizar notificaciones o tareas similares.

Los scripts del lado cliente relacionados con la confirmación de cambios pueden ser utilizados en prácticamente cualquier flujo de trabajo. A menudo, se suelen utilizar para obligar a seguir ciertas reglas; aunque es importante indicar que estos scripts no se transfieren durante el clonado. Puedes implantar reglas en el lado servidor para rechazar envíos (push) que no cumplan ciertos estándares, pero es completamente voluntario para los desarrolladores el utilizar scripts en el lado cliente. Por tanto, estos scripts son para ayudar a los desarrolladores, y, como tales, han de ser

configurados y mantenidos por ellos, pudiendo ser sobrescritos o modificados por ellos en cualquier momento.

### **Puntos en el flujo de trabajo del correo electrónico**

Tienes disponibles tres puntos de enganche en el lado cliente para interactuar con el flujo de trabajo de correo electrónico. Todos ellos se invocan al utilizar el comando 'git am', por lo que si no utilizas dicho comando, puedes saltar directamente a la siguiente sección. Si recibes parches a través de correo-e preparados con 'git format-patch', es posible que parte de lo descrito en esta sección te pueda ser útil.

El primer punto de enganche que se activa es 'applypatch-msg'. Recibe un solo argumento: el nombre del archivo temporal que contiene el mensaje de confirmación propuesto. Git abortará la aplicación del parche si este script termina con un código de salida distinto de cero. Puedes utilizarlo para asegurarte de que el mensaje de confirmación esté correctamente formateado o para normalizar el mensaje permitiendo al script que lo edite sobre la marcha.

El siguiente punto de enganche que se activa al aplicar parches con 'git am' es el punto 'pre-applypatch'. No recibe ningún argumento de entrada y se lanza después de que el parche haya sido aplicado, por lo que puedes utilizarlo para revisar la situación (snapshot) antes de confirmarla. Con este script, puedes lanzar pruebas o similares para chequear el árbol de trabajo. Si falta algo o si alguna de las pruebas falla, saliendo con un código de salida distinto de cero abortará el comando 'git am' sin confirmar el parche.

El último punto de enganche que se activa durante una operación 'git am' es el punto 'post-applypatch'. Puedes utilizarlo para notificar de su aplicación al grupo o al autor del parche. No puedes detener el proceso de parcheo con este script.

### **Otros puntos de enganche del lado cliente**

El punto 'pre-rebase' se activa antes de cualquier reorganización y puede abortarla si retorna con un código de salida distinto de cero. Puedes usarlo para impedir reorganizaciones de cualquier confirmación de cambios ya enviada (push) a algún servidor. El script de ejemplo para 'pre-rebase' hace precisamente eso, aunque asumiendo que 'next' es el nombre de la rama publicada. Si lo vas a utilizar, tendrás que modificarlo para que se ajuste al nombre que tenga tu rama publicada.

Tras completarse la ejecución de un comando 'git checkout', es cuando se activa el punto de enganche 'post-checkout'. Lo puedes utilizar para ajustar tu carpeta de trabajo al entorno de tu

proyecto. Entre otras cosas, puedes mover grandes archivos binarios de los que no quieras llevar control, puedes autogenerar documentación,....

Y, por último, el punto de enganche 'post-merge' se activa tras completarse la ejecución de un comando 'git merge'. Puedes utilizarlo para recuperar datos de tu carpeta de trabajo que Git no puede controlar, como por ejemplo datos relativos a permisos. Este punto de enganche puede utilizarse también para comprobar la presencia de ciertos archivos, externos al control de Git, que desees copiar cada vez que cambie la carpeta de trabajo.

## **Puntos de enganche del lado servidor**

Aparte de los puntos del lado cliente, como administrador de sistemas, puedes utilizar un par de puntos de enganche importantes en el lado servidor; para implementar prácticamente cualquier tipo de política que quieras mantener en tu proyecto. Estos scripts se lanzan antes y después de cada envío (push) al servidor. El script previo, puede terminar con un código de salida distinto de cero y abortar el envío, devolviendo el correspondiente mensaje de error al cliente. Este script puede implementar políticas de recepción tan complejas como desees.

### **'pre-receive' y 'post-receive'**

El primer script que se activa al manejar un envío de un cliente es el correspondiente al punto de enganche 'pre-receive'. Recibe una lista de referencias que se están enviando (push) desde la entrada estandar (stdin); y, si termina con un código de salida distinto de cero, ninguna de ellas será aceptada. Puedes utilizar este punto de enganche para realizar tareas tales como la de comprobar que ninguna de las referencias actualizadas no son de avance directo (non-fast-forward); o para comprobar que el usuario que realiza el envío tiene realmente permisos para crear, borrar o modificar cualquiera de los archivos que está tratando de cambiar.

El punto de enganche 'post-receive' se activa cuando termina todo el proceso, y se puede utilizar para actualizar otros servicios o para enviar notificaciones a otros usuarios. Recibe los mismos datos que 'pre-receive' desde la entrada estandar. Algunos ejemplos de posibles aplicaciones pueden ser la de alimentar una lista de correo-e, avisar a un servidor de integración continua, o actualizar un sistema de seguimiento de tickets de servicio --pudiendo incluso procesar el mensaje de confirmación para ver si hemos de abrir, modificar o dar por cerrado algún ticket--. Este script no puede detener el proceso de envío, pero el cliente no se desconecta hasta que no se completa su ejecución; por tanto, has de ser cuidadoso cuando intentes realizar con él tareas que puedan requerir mucho tiempo.

### **update**

El punto de enganche 'update' es muy similar a 'pre-receive', pero con la diferencia de que se activa una vez por cada rama que se está intentando actualizar con el envío. Si la persona que realiza el envío intenta actualizar varias ramas, 'pre-receive' se ejecuta una sola vez, mientras que 'update' se ejecuta tantas veces como ramas se estén actualizando. El lugar de recibir datos desde la entrada estandar (stdin), este script recibe tres argumentos: el nombre de la rama, la clave SHA-1 a la que esta apuntada antes del envío, y la clave SHA-1 que el usuario está intentando enviar. Si el script 'update' termina con un código de salida distinto de cero, únicamente los cambios de esa rama son rechazados; el resto de ramas continuarán con sus actualizaciones.

## 7.4 Personalizando Git - Un ejemplo de implantación de una determinada política en Git

### Un ejemplo de implantación de una determinada política en Git

En esta sección, utilizarás lo aprendido para establecer un flujo de trabajo en Git que: compruebe si los mensajes de confirmación de cambios encajan en un determinado formato, obligue a realizar solo envíos de avance directo, y permita solo a ciertos usuarios modificar ciertas carpetas del proyecto. Para ello, has de preparar los correspondientes scripts de cliente (para ayudar a los desarrolladores a saber de antemano si sus envíos van a ser rechazados o no), y los correspondientes scripts de servidor (para obligar a cumplir esas políticas).

He usado Ruby para escribir los ejemplos, tanto porque es mi lenguaje preferido de scripting y porque creo que es el más parecido a pseudocódigo; de tal forma que puedas ser capaz de seguir el código, incluso si no conoces Ruby. Pero, puede ser igualmente válido cualquier otro lenguaje. Todos los scripts de ejemplo que vienen de serie con Git están escritos en Perl o en Bash shell, por lo que tienes bastantes ejemplos en esos lenguajes de scripting.

#### Punto de enganche en el lado servidor

Todo el trabajo del lado servidor va en el script 'update' de la carpeta 'hooks'. El script 'update' se lanza una vez por cada rama que se envía (push) al servidor; y recibe la referencia de la rama a la que se envía, la antigua revisión en que estaba la rama y la nueva revisión que se está enviando. También puedes tener acceso al usuario que está enviando, si este los envía a través de SSH. Si has permitido a cualquiera conectarse con un mismo usuario (como "git", por ejemplo), has

tenido que dar a dicho usuario una envoltura (shell wrapper) que te permite determinar cual es el usuario que se conecta según sea su clave pública, permitiéndote fijar una variable de entorno especificando dicho usuario. Aquí, asumiremos que el usuario conectado queda reflejado en la variable de entorno '\$USER', de tal forma que el script 'update' comienza recogiendo toda la información que necesitas:

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

```
$refname = ARGV[0]
$oldrev  = ARGV[1]
$newrev  = ARGV[2]
$user    = ENV['USER']
```

```
puts "Enforcing Policies... \n(#{ $refname }) (#{ $oldrev[0,6] }) (#{ $newrev[0,6] })"
```

Sí, estoy usando variables globales. No me juzgues por ello, --es más sencillo mostrarlo de esta manera--.

### **Obligando a utilizar un formato específico en el mensaje de confirmación de cambios**

Tu primer reto es asegurarte que todos y cada uno de los mensajes de confirmación de cambios se ajustan a un determinado formato. Simplemente por fijar algo concreto, supongamos que cada mensaje ha de incluir un texto tal como "ref: 1234", porque quieres enlazar cada confirmación de cambios con una determinada entrada de trabajo en un sistema de control. Has de mirar en cada confirmación de cambios (commit) recibida, para ver si contiene ese texto; y, si no lo trae, salir con un código distinto de cero, de tal forma que el envío (push) sea rechazado.

Puedes obtener la lista de las claves SHA-1 de todos las confirmaciones de cambios enviadas cogiendo los valores de '\$newrev' y de '\$oldrev', y pasandolos a comando de mantenimiento de Git llamado 'git rev-list'. Este comando es básicamente el mismo que 'git log', pero por defecto, imprime solo los valores SHA-1 y nada más. Con él, puedes obtener la lista de todas las claves SHA que se han introducido entre una clave SHA y otra clave SHA dadas; obtendrás algo así como esto:

```
$ git rev-list 538c33..d14fc7
d14fc7c847ab946ec39590d87783c69b031bdfb7
9f585da4401b0a3999e84113824d15245c13f0be
234071a1be950e2a8d078e6141f5cd20c1e61ad3
dfa04c9ef3d5197182f13fb5b9b1fb7717d2222a
```

```
17716ec0f1ff5c77eff40b7fe912f9f6cfd0e475
```

Puedes coger esta salida, establecer un bucle para recorrer cada una de esas confirmaciones de cambios, coger el mensaje de cada una y comprobarlo contra una expresión regular de búsqueda del patrón deseado.

Tienes que imaginarte cómo puedes obtener el mensaje de cada una de esas confirmaciones de cambios a comprobar. Para obtener los datos "en crudo" de una confirmación de cambios, puedes utilizar otro comando de mantenimiento de Git denominado 'git cat-file'. En el capítulo 9 volveremos en detalle sobre estos comandos de mantenimiento; pero, por ahora, esto es lo que obtienes con dicho comando:

```
$ git cat-file commit ca82a6
tree cfd3bf379e4f8dba8717dee55aab78aef7f4daf
parent 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
author Scott Chacon <schacon@gmail.com> 1205815931 -0700
committer Scott Chacon <schacon@gmail.com> 1240030591 -0700
```

```
changed the version number
```

Una vía sencilla para obtener el mensaje, es la de ir hasta la primera línea en blanco y luego coger todo lo que siga a esta. En los sistemas Unix, lo puedes realizar con el comando 'sed':

```
$ git cat-file commit ca82a6 | sed '1,/^\$/d'
changed the version number
```

Puedes usar este "hechizo mágico" para coger el mensaje de cada confirmación de cambios que se está enviando y salir si localizas algo que no cuadra en alguno de ellos. Para salir del script y rechazar el envío, recuerda que debes salir con un código distinto de cero. El método completo será algo así como:

```
$regex = /\[ref: (\d+)\]/
$regex = /\[ref: (\d+)\]/

# enforced custom commit message format
def check_message_format
  missed_revs = `git rev-list #{$oldrev}..#{$newrev}`.split("\n")
  missed_revs.each do |rev|
    message = `git cat-file commit #{rev} | sed '1,/^\$/d'`
    if !$regex.match(message)
```

```
puts "[POLICY] Your message is not formatted correctly"
exit 1
end
end
end
check_message_format
```

Poniendo esto en tu script 'update', serán rechazadas todas las actualizaciones que contengan cambios con mensajes que no se ajusten a tus reglas.

### **Implementando un sistema de control de accesos basado en usuario**

Imaginemos que deseas implementar un sistema de control de accesos (Access Control List, ACL). Para vigilar qué usuarios pueden enviar (push) cambios a qué partes de tus proyectos. Algunas personas tendrán acceso completo, y otras tan solo acceso a ciertas carpetas o a ciertos archivos. Para implementar esto, has de escribir esas reglas de acceso en un archivo denominado 'acl' ubicado en tu repositorio git básico en el servidor. Y tienes que preparar el enganche 'update' para hacerle consultar esas reglas, mirar los archivos que están siendo subidos en las confirmaciones de cambio (commit) enviadas (push), y determinar así si el usuario emisor del envío tiene o no permiso para actualizar esos archivos.

Como hemos dicho, el primer paso es escribir tu lista de control de accesos (ACL). Su formato es muy parecido al del mecanismo CVS ACL: utiliza una serie de líneas donde el primer campo es 'avail' o 'unavail' (permitido o no permitido), el segundo campo es una lista de usuarios separados por comas, y el último campo es la ubicación (path) sobre el que aplicar la regla (dejarlo en blanco equivale a un acceso abierto). Cada uno de esos campos se separan entre sí con el caracter barra vertical ('|').

Por ejemplo, si tienes un par de administradores, algunos redactores técnicos con acceso a la carpeta 'doc', y un desarrollador que únicamente accede a las carpetas 'lib' y 'test', el archivo ACL resultante sería:

```
avail|nickh,pjhyett,defunkt,tpw
avail|usinclair,cdickens,ebronte|doc
avail|schacon|lib
avail|schacon|tests
```

Para implementarlo, hemos de leer previamente estos datos en una estructura que podamos emplear. En este caso, por razones de simplicidad, vamos a mostrar únicamente la forma de



implementar las directivas 'avail' (permitir). Este es un método que te devuelve un array asociativo cuya clave es el nombre del usuario y su valor es un array de ubicaciones (paths) donde ese usuario tiene acceso de escritura:

```
def get_acl_access_data(acl_file)
  # read in ACL data
  acl_file = File.read(acl_file).split("\n").reject { |line| line == " }
  access = {}
  acl_file.each do |line|
    avail, users, path = line.split('|')
    next unless avail == 'avail'
    users.split(',').each do |user|
      access[user] ||= []
      access[user] << path
    end
  end
  access
end
```

Si lo aplicamos sobre la lista ACL descrita anteriormente, este método 'get acl access\_data' devolverá una estructura de datos similar a esta:

```
{"defunkt"=>[nil],
"tpw"=>[nil],
"nickh"=>[nil],
"pjhyett"=>[nil],
"schacon"=>["lib", "tests"],
"cdickens"=>["doc"],
"usinclair"=>["doc"],
"ebronte"=>["doc"]}
```

Una vez tienes los permisos en orden, necesitas averiguar las ubicaciones modificadas por las confirmaciones de cambios enviadas; de tal forma que puedas asegurarte de que el usuario que las está enviando tiene realmente permiso para modificarlas.

Puedes comprobar fácilmente qué archivos han sido modificados en cada confirmación de cambios, utilizando la opción '--name-only' del comando 'git log' (citado brevemente en el capítulo 2):

```
$ git log -1 --name-only --pretty=format:" %f585d
```

## README

### lib/test.rb

Utilizando la estructura ACL devuelta por el método 'getaclaccess\_data' y comprobandola sobre la lista de archivos de cada confirmación de cambios, puedes determinar si el usuario tiene o no permiso para enviar dichos cambios:

```
# only allows certain users to modify certain subdirectories in a project
def check_directory_perms
  access = get_acl_access_data('acl')

  # see if anyone is trying to push something they can't
  new_commits = `git rev-list #{oldrev}..#{newrev}`.split("\n")
  new_commits.each do |rev|
    files_modified = `git log -1 --name-only --pretty=format:" #{rev}"`.split("\n")
    files_modified.each do |path|
      next if path.size == 0
      has_file_access = false
      access[$user].each do |access_path|
        if !access_path || # user has access to everything
          (path.index(access_path) == 0) # access to this path
          has_file_access = true
        end
      end
      if !has_file_access
        puts "[POLICY] You do not have access to push to #{path}"
        exit 1
      end
    end
  end
end

check_directory_permscheck_directory_perms
```

La mayor parte de este código debería de ser sencillo de leer. Con 'git rev-list', obtienes una lista de las nuevas confirmaciones de cambio enviadas a tu servidor. Luego, para cada una de ellas, localizas los archivos modificados y te aseguras de que el usuario que las envia tiene realmente acceso a todas las ubicaciones que pretende modificar. Un "rubysmo" que posiblemente sea un tanto oscuro puede ser 'path.index(accesspath) == 0'. *Simplemente devuelve verdadero en el*

*caso de que la ubicacion comience por 'accesspath' ; de esta forma, nos aseguramos de que 'access\_path' no esté solo contenido en una de las ubicaciones permitidas, sino sea una ubicación permitida la que comience con la ubicación accedida.*

Una vez implementado todo esto, tus usuarios no podrán enviar confirmaciones de cambios con mensajes mal formados o con modificaciones sobre archivos fuera de las ubicaciones que les hayas designado.

### **Obligando a realizar envíos solo-de-avance-rápido (Fast-Forward-Only pushes)**

Lo único que nos queda por implementar es un mecanismo para limitar los envíos a envíos de avance rápido (Fast-Forward-Only pushes). En las versiones a partir de la 1.6, puedes ajustar las opciones 'receive.denyDeletes' (prohibir borrados) y 'receive.denyNonFastForwards' (prohibir envíos que no sean avances-rápidos). Pero haciéndolo a través de un enganche (hook), podrá funcionar también en versiones anteriores de Git, podrás modificarlo para que actúe únicamente sobre ciertos usuarios, o podrás realizar cualquier otra acción que estimes oportuna.

La lógica para hacer así la comprobación es la de mirar por si alguna confirmación de cambios se puede alcanzar desde la versión más antigua pero no desde la más reciente. Si hay alguna, entonces es un envío de avance-rápido (fast-forward push); sino hay ninguna, es un envío a prohibir:

```
# enforces fast-forward only pushes
def check_fast_forward
  missed_refs = `git rev-list #{$newrev}..#{$oldrev}`
  missed_ref_count = missed_refs.split("\n").size
  if missed_ref_count > 0
    puts "[POLICY] Cannot push a non fast-forward reference"
    exit 1
  end
end

check_fast_forward
```

Una vez esté todo listo. Si lanzas el comando 'chmod u+x .git/hooks/update', siendo este el archivo donde has puesto todo este código; y luego intentas enviar una referencia que no sea de avance-rápido, obtendrás algo como esto:

```
$ git push -f origin master
Counting objects: 5, done.
Compressing objects: 100% (3/3), done.
```

```
Writing objects: 100% (3/3), 323 bytes, done.
Total 3 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
Enforcing Policies...
(refs/heads/master) (8338c5) (c5b616)
[POLICY] Cannot push a non-fast-forward reference
error: hooks/update exited with error code 1
error: hook declined to update refs/heads/master
To git@gitserver:project.git
! [remote rejected] master -> master (hook declined)
error: failed to push some refs to 'git@gitserver:project.git'[remote rejected] master
-> master (hook declined)
error: failed to push some refs to 'git@gitserver:project.git'
```

Tenemos un para de aspectos interesantes aquí. Lo primero observado cuando el enganche (hook) arranca es:

```
Enforcing Policies...
(refs/heads/master) (fb8c72) (c56860)
```

Precisamente, lo enviado a la salida estandar stdout justo al principio del script de actualización. Cabe destacar que todo lo que se envíe a la salida estandar stdout, será transferido al cliente.

Lo segundo que se puede apreciar es el mensaje de error:

```
[POLICY] Cannot push a non fast-forward reference
error: hooks/update exited with error code 1
error: hook declined to update refs/heads/master
```

La primera línea la has enviado tú, pero las otras dos son de Git. Indicando que el script de actualización ha terminado con código no-cero y, por tanto, ha rechazado la modificación. Y, por último, se ve:

```
To git@gitserver:project.git
! [remote rejected] master -> master (hook declined)
error: failed to push some refs to 'git@gitserver:project.git'[remote rejected] master
-> master (hook declined)
error: failed to push some refs to 'git@gitserver:project.git'
```

Un mensaje por cada referencia rechazada por el enganche (hook) de actualización, especificando que ha sido rechazada precisamente por un fallo en el enganche.

Es más, si la referencia (ref marker) no se encuentra presente para alguna de las confirmaciones de cambio, verás el mensaje de error previsto para ello:

[POLICY] Your message is not formatted correctly

O si alguien intenta editar un archivo sobre el que no tiene acceso y luego envía una confirmación de cambios con ello, verá también algo similar. Por ejemplo, si un editor técnico intenta enviar una confirmación de cambios donde se haya modificado algo de la carpeta 'lib', verá:

[POLICY] You do not have access to push to lib/test.rb

Y eso es todo. De ahora en adelante, en tanto en cuando el script 'update' este presente y sea ejecutable, tu repositorio nunca se verá perjudicado, nunca tendrá un mensaje de confirmación de cambios sin tu plantilla y tus usuarios estarán controlados.

## **Puntos de enganche del lado cliente**

Lo malo del sistema descrito en la sección anterior pueden ser los lamentos que inevitablemente se van a producir cuando los envíos de tus usuarios sean rechazados. Ver rechazado en el último minuto su tan cuidadosamente preparado trabajo, puede ser realmente frustrante. Y, aún peor, tener que reescribir su histórico para corregirlo puede ser un auténtico calvario.

La solución a este dilema es el proporcionarles algunos enganches (hook) del lado cliente, para que les avisen cuando están trabajando en algo que el servidor va a rechazarles. De esta forma, pueden corregir los problemas antes de confirmar cambios y antes de que se conviertan en algo realmente complicado de arreglar. Debido a que estos enganches no se transfieren junto con el clonado de un proyecto, tendrás que distribuirlos de alguna otra manera. Y luego pedir a tus usuarios que se los copien a sus carpetas '.git/hooks' y los hagan ejecutables. Puedes distribuir esos enganches dentro del mismo proyecto o en un proyecto separado. Pero no hay modo de implementarlos automáticamente.

Para empezar, se necesita chequear el mensaje de confirmación inmediatamente antes de cada confirmación de cambios, para asegurarse de que el servidor no los rechazará debido a un mensaje mal formateado. Para ello, se añade el enganche 'commit-msg'. Comparando el mensaje del

archivo pasado como primer argumento con el mensaje patrón, puedes obligar a Git a abortar la confirmación de cambios (commit) en caso de no coincidir ambos:

```
#!/usr/bin/env ruby
message_file = ARGV[0]
message = File.read(message_file)

$regex = /\[ref: (\d+)\]/$regex = /\[ref: (\d+)\]/

if !$regex.match(message)
  puts "[POLICY] Your message is not formatted correctly"
  exit 1
end
```

Si este script está en su sitio (el archivo '.git/hooks/commit-msg') y es ejecutable, al confirmar cambios con un mensaje inapropiado, verás algo así como:

```
$ git commit -am 'test'
[POLICY] Your message is not formatted correctly
```

Y la confirmación no se llevará a cabo. Sin embargo, si el mensaje está formateado adecuadamente, Git te permitirá confirmar cambios:

```
$ git commit -am 'test [ref: 132]'
[master e05c914] test [ref: 132]
1 files changed, 1 insertions(+), 0 deletions(-)
```

A continuación, se necesita también asegurarse de no estar modificando archivos fuera del alcance de tus permisos. Si la carpeta '.git' de tu proyecto contiene una copia del archivo de control de accesos (ACL) utilizada previamente, este script 'pre-commit' podrá comprobar los límites:

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

$user = ENV['USER']

# [ insert acl_access_data method from above ]

# only allows certain users to modify certain subdirectories in a project
def check_directory_perms
```

```

access = get_acl_access_data('.git/acl')

files_modified = `git diff-index --cached --name-only HEAD`.split("\n")
files_modified.each do |path|
  next if path.size == 0
  has_file_access = false
  access[$user].each do |access_path|
    if !access_path || (path.index(access_path) == 0)
      has_file_access = true
    end
  end
  if !has_file_access
    puts "[POLICY] You do not have access to push to #{path}"
    exit 1
  end
end
end

check_directory_permscheck_directory_perms

```

Este es un script prácticamente igual al del lado servidor. Pero con dos importantes diferencias. La primera es que el archivo ACL está en otra ubicación, debido a que el script corre desde tu carpeta de trabajo y no desde la carpeta de Git. Esto obliga a cambiar la ubicación del archivo ACL de

```

access = get_acl_access_data('acl')

a

access = get_acl_access_data('.git/acl')

```

La segunda diferencia es la forma de listar los archivos modificados. Debido a que el metodo del lado servidor utiliza el registro de confirmaciones de cambio, pero, sin embargo, aquí la confirmación no se ha registrado aún, la lista de archivos se ha de obtener desde el área de preparación (staging area). En lugar de

```
files_modified = `git log -1 --name-only --pretty=format:" #{ref}`
```

tenemos que utilizar

```
files_modified = `git diff-index --cached --name-only HEAD`
```

Estas dos son las únicas diferencias; en todo lo demás, el script funciona de la misma manera. Es necesario advertir de que se espera que trabajes localmente con el mismo usuario con el que enviarás (push) a la máquina remota. Si no fuera así, tendrás que ajustar manualmente la variable '\$user'.

El último aspecto a comprobar es el de no intentar enviar referencias que no sean de avance-rápido. Pero esto es algo más raro que suceda. Para tener una referencia que no sea de avance-rápido, tienes que haber reorganizado (rebase) una confirmación de cambios (commit) ya enviada anteriormente, o tienes que estar tratando de enviar una rama local distinta sobre la misma rama remota.

De todas formas, el único aspecto accidental que puede interesante capturar son los intentos de reorganizar confirmaciones de cambios ya enviadas. El servidor te avisará de que no puedes enviar ningún no-avance-rápido, y el enganche te impedirá cualquier envío forzado

Este es un ejemplo de script previo a reorganización que lo puede comprobar. Con la lista de confirmaciones de cambio que estás a punto de reescribir, las comprueba por si alguna de ellas existe en alguna de tus referencias remotas. Si encuentra alguna, aborta la reorganización:

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

```
base_branch = ARGV[0]
if ARGV[1]
  topic_branch = ARGV[1]
else
  topic_branch = "HEAD"
end
```

```
target_shas = `git rev-list #{base_branch}..#{topic_branch}`.split("\n")
remote_refs = `git branch -r`.split("\n").map { |r| r.strip }
```

```
target_shas.each do |sha|
  remote_refs.each do |remote_ref|
    shas_pushed = `git rev-list ^#{sha}^@ refs/remotes/#{remote_ref}`
    if shas_pushed.split("\n").include?(sha)
      puts "[POLICY] Commit #{sha} has already been pushed to #{remote_ref}"
      exit 1
    end
  end
end
```



```
end  
end
```

Este script utiliza una sintaxis no contemplada en la sección de Selección de Revisiones del capítulo 6. La lista de confirmaciones de cambio previamente enviadas, se comprueba con:

```
git rev-list ^#{sha}^@ refs/remotes/#{remote_ref}
```

La sintaxis 'SHA^@' recupera todos los padres de esa confirmación de cambios (commit). Estas mirando por cualquier confirmación que se pueda alcanzar desde la última en la parte remota, pero que no se pueda alcanzar desde ninguno de los padres de cualquiera de las claves SHA que estás intentando enviar. Es decir, confirmaciones de avance-rápido.

La mayor pega de este sistema es el que puede llegar a ser muy lento; y muchas veces es innecesario, ya que el propio servidor te va a avisar y te impedirá el envío, siempre y cuando no intentes forzar dicho envío con la opción '-f'. De todas formas, es un ejercicio interesante. Y, en teoría al menos, pude ayudarte a evitar reorganizaciones que luego tengas de hechar para atrás y arreglarlas.

# Chapter 8

## Git y Otros Sistemas

El mundo no es perfecto. Lo más normal es que no puedas cambiar inmediatamente a Git cada proyecto que te encuentras. Algunas veces estás atascado en un proyecto utilizando otro VCS, y muchas veces ese sistema es Subversion. Pasaremos la primera parte de este capítulo aprendiendo sobre **git svn**, la puerta de enlace de Subversion bidireccional en Git.

En un momento dado, quizá quieras convertir tu proyecto existente a Git. La segunda parte de este capítulo cubre cómo migrar tu proyecto a Git: primero desde Subversion, luego desde Perforce, y finalmente por medio de un script de importación a medida para casos no estándar.

## 8.1 Git y Otros Sistemas - Git y Subversion

### Git y Subversion

Hoy por hoy, la mayoría de los proyectos de código abierto y un gran número de proyectos corporativos usan Subversion para manejar su código fuente. Es el VCS más popular y lleva ahí casi una década. También es muy similar en muchas cosas a CVS, que fue el rey del control de código fuente anteriormente.

Una de las grandes características de Git es el puente bidireccional llamado **git svn**. Esta herramienta permite el uso de Git como un cliente válido para un servidor Subversion, así que puedes utilizar todas las características locales de Git y luego hacer publicaciones al servidor de Subversion como si estuvieras usando Subversion localmente. Esto significa que puedes ramificar y fusionar localmente, usar el área de preparación, reconstruir, entresacar, etc., mientras tus colaboradores continúan usando sus antiguos y oscuros métodos. Es una buena forma de colar a Git dentro de un ambiente corporativo y ayudar a tus colegas desarrolladores a hacerse más eficientes mientras tu haces presión para que se cambie la infraestructura y el soporte de Git sea completo. El puente de Subversion es la "droga de entrada" al mundo de DVCS.

[git svn](#)

El comando básico de Git para todos los comandos de enlace con Subversion es **git svn**. Siempre debes empezar con eso. Hay unos cuantos, por lo que vamos a aprender los básicos recorriendo unos pocos flujos de trabajo pequeños.

Es importante fijarse en que cuando usas **git svn** estás interactuando con Subversion, que es un sistema mucho menos sofisticado que Git. Aunque puedes hacer ramas y fusiones localmente, lo mejor es mantener tu historia lo más lineal posible mediante reorganizaciones, y evitar hacer cosas como interactuar a la vez con un repositorio remoto de Git.

No reescribas tu historia y trates de publicar de nuevo, y no publiques a un repositorio Git paralelo para colaborar con colegas que usen Git al mismo tiempo. Subversion sólo puede tener una historia lineal simple, de lo contrario puedes confundirlo fácilmente. Si trabajas con un equipo, y algunos usan SVN mientras otros usan Git, asegúrate de que todos usan el servidor de SVN para colaborar. Si lo haces así, tu vida será más simple.

## Setting Up

To demonstrate this functionality, you need a typical SVN repository that you have write access to. If you want to copy these examples, you'll have to make a writeable copy of my test repository. In order to do that easily, you can use a tool called **svnsync** that comes with more recent versions of Subversion — it should be distributed with at least 1.4. For these tests, I created a new Subversion repository on Google code that was a partial copy of the **protobuf** project, which is a tool that encodes structured data for network transmission.

To follow along, you first need to create a new local Subversion repository:

```
$ mkdir /tmp/test-svn
$ svnadmin create /tmp/test-svn
```

Then, enable all users to change revprops — the easy way is to add a pre-revprop-change script that always exits 0:

```
$ cat /tmp/test-svn/hooks/pre-revprop-change
#!/bin/sh
exit 0;
$ chmod +x /tmp/test-svn/hooks/pre-revprop-change
```

You can now sync this project to your local machine by calling **svnsync init** with the to and from repositories.

```
$ svnsync init file:///tmp/test-svn http://progit-example.googlecode.com/svn/
```

This sets up the properties to run the sync. You can then clone the code by running

```
$ svnsync sync file:///tmp/test-svn
Committed revision 1.
Copied properties for revision 1.
Committed revision 2.
Copied properties for revision 2.
Committed revision 3.
...
```

Although this operation may take only a few minutes, if you try to copy the original repository to another remote repository instead of a local one, the process will take nearly an hour, even though there are fewer than 100 commits. Subversion has to clone one revision at a time and then push it back into another repository — it's ridiculously inefficient, but it's the only easy way to do this.

## Getting Started

Now that you have a Subversion repository to which you have write access, you can go through a typical workflow. You'll start with the `git svn clone` command, which imports an entire Subversion repository into a local Git repository. Remember that if you're importing from a real hosted Subversion repository, you should replace the `file:///tmp/test-svn` here with the URL of your Subversion repository:

```
$ git svn clone file:///tmp/test-svn -T trunk -b branches -t tags
Initialized empty Git repository in /Users/schacon/projects/testsvnsync/svn/.git/
r1 = b4e387bc68740b5af56c2a5faf4003ae42bd135c (trunk)
  A   m4/acx_pthread.m4
  A   m4/stl_hash.m4
...
r75 = d1957f3b307922124eec6314e15bcda59e3d9610 (trunk)
Found possible branch point: file:///tmp/test-svn/trunk => \
  file:///tmp/test-svn /branches/my-calc-branch, 75
Found branch parent: (my-calc-branch)
d1957f3b307922124eec6314e15bcda59e3d9610
Following parent with do_switch
Successfully followed parent
r76 = 8624824ecc0badd73f40ea2f01fce51894189b01 (my-calc-branch)
```

Checked out HEAD:

file:///tmp/test-svn/branches/my-calc-branch r76

This runs the equivalent of two commands — `git svn init` followed by `git svn fetch` — on the URL you provide. This can take a while. The test project has only about 75 commits and the codebase isn't that big, so it takes just a few minutes. However, Git has to check out each version, one at a time, and commit it individually. For a project with hundreds or thousands of commits, this can literally take hours or even days to finish.

The `-T trunk -b branches -t tags` part tells Git that this Subversion repository follows the basic branching and tagging conventions. If you name your trunk, branches, or tags differently, you can change these options. Because this is so common, you can replace this entire part with `-s`, which means standard layout and implies all those options. The following command is equivalent:

```
$ git svn clone file:///tmp/test-svn -s
```

At this point, you should have a valid Git repository that has imported your branches and tags:

```
$ git branch -a
* master
  my-calc-branch
  tags/2.0.2
  tags/release-2.0.1
  tags/release-2.0.2
  tags/release-2.0.2rc1
  trunk
```

It's important to note how this tool namespaces your remote references differently. When you're cloning a normal Git repository, you get all the branches on that remote server available locally as something like `origin/[branch]` - namespaced by the name of the remote. However, `git svn` assumes that you won't have multiple remotes and saves all its references to points on the remote server with no namespacing. You can use the Git plumbing command `show-ref` to look at all your full reference names:

```
$ git show-ref
1cbd4904d9982f386d87f88fce1c24ad7c0f0471 refs/heads/master
aee1ecc26318164f355a883f5d99cff0c852d3c4 refs/remotes/my-calc-branch
03d09b0e2aad427e34a6d50ff147128e76c0e0f5 refs/remotes/tags/2.0.2
```

```
50d02cc0adc9da4319eeba0900430ba219b9c376 refs/remotes/tags/release-2.0.1
4caaa711a50c77879a91b8b90380060f672745cb refs/remotes/tags/release-2.0.2
1c4cb508144c513ff1214c3488abe66dcb92916f refs/remotes/tags/release-2.0.2rc1
1cbd4904d9982f386d87f88fce1c24ad7c0f0471 refs/remotes/trunk
```

A normal Git repository looks more like this:

```
$ git show-ref
83e38c7a0af325a9722f2fdc56b10188806d83a1 refs/heads/master
3e15e38c198baac84223acfc6224bb8b99ff2281 refs/remotes/gitserver/master
0a30dd3b0c795b80212ae723640d4e5d48cabdff refs/remotes/origin/master
25812380387fdd55f916652be4881c6f11600d6f refs/remotes/origin/testing
```

You have two remote servers: one named **gitserver** with a **master** branch; and another named **origin** with two branches, **master** and **testing**.

Notice how in the example of remote references imported from **git svn**, tags are added as remote branches, not as real Git tags. Your Subversion import looks like it has a remote named tags with branches under it.

## Committing Back to Subversion

Now that you have a working repository, you can do some work on the project and push your commits back upstream, using Git effectively as a SVN client. If you edit one of the files and commit it, you have a commit that exists in Git locally that doesn't exist on the Subversion server:

```
$ git commit -am 'Adding git-svn instructions to the README'
[master 97031e5] Adding git-svn instructions to the README
1 files changed, 1 insertions(+), 1 deletions(-)
```

Next, you need to push your change upstream. Notice how this changes the way you work with Subversion — you can do several commits offline and then push them all at once to the Subversion server. To push to a Subversion server, you run the **git svn dcommit** command:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
    M    README.txt
Committed r79
    M    README.txt
r79 = 938b1a547c2cc92033b74d32030e86468294a5c8 (trunk)
```

No changes between current HEAD and refs/remotes/trunk  
Resetting to the latest refs/remotes/trunk

This takes all the commits you've made on top of the Subversion server code, does a Subversion commit for each, and then rewrites your local Git commit to include a unique identifier. This is important because it means that all the SHA-1 checksums for your commits change. Partly for this reason, working with Git-based remote versions of your projects concurrently with a Subversion server isn't a good idea. If you look at the last commit, you can see the new `git-svn-id` that was added:

```
$ git log -1
commit 938b1a547c2cc92033b74d32030e86468294a5c8
Author: schacon <schacon@4c93b258-373f-11de-be05-5f7a86268029>
Date: Sat May 2 22:06:44 2009 +0000
```

Adding git-svn instructions to the README

```
git-svn-id: file:///tmp/test-svn/trunk@79
4c93b258-373f-11de-be05-5f7a86268029
```

Notice that the SHA checksum that originally started with `97031e5` when you committed now begins with `938b1a5`. If you want to push to both a Git server and a Subversion server, you have to push (`dcommit`) to the Subversion server first, because that action changes your commit data.

## Pulling in New Changes

If you're working with other developers, then at some point one of you will push, and then the other one will try to push a change that conflicts. That change will be rejected until you merge in their work. In `git svn`, it looks like this:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
Merge conflict during commit: Your file or directory 'README.txt' is probably \
out-of-date: resource out of date; try updating at /Users/schacon/libexec/git-\
core/git-svn line 482
```

To resolve this situation, you can run `git svn rebase`, which pulls down any changes on the server that you don't have yet and rebases any work you have on top of what is on the server:

```
$ git svn rebase
M    README.txt
r80 = ff829ab914e8775c7c025d741beb3d523ee30bc4 (trunk)
First, rewinding head to replay your work on top of it...
Applying: first user change
```

Now, all your work is on top of what is on the Subversion server, so you can successfully dcommit:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
M    README.txt
Committed r81
M    README.txt
r81 = 456cbe6337abe49154db70106d1836bc1332deed (trunk)
No changes between current HEAD and refs/remotes/trunk
Resetting to the latest refs/remotes/trunk
```

It's important to remember that unlike Git, which requires you to merge upstream work you don't yet have locally before you can push, **git svn** makes you do that only if the changes conflict. If someone else pushes a change to one file and then you push a change to another file, your **dcommit** will work fine:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
M    configure.ac
Committed r84
M    autogen.sh
r83 = 8aa54a74d452f82eee10076ab2584c1fc424853b (trunk)
M    configure.ac
r84 = cdbac939211ccb18aa744e581e46563af5d962d0 (trunk)
W: d2f23b80f67aaaa1f6f5aaef48fce3263ac71a92 and refs/remotes/trunk differ, \
  using rebase:
:100755 100755 efa5a59965fbbb5b2b0a12890f1b351bb5493c18 \
  015e4c98c482f0fa71e4d5434338014530b37fa6 M    autogen.sh
First, rewinding head to replay your work on top of it...
Nothing to do.
```



This is important to remember, because the outcome is a project state that didn't exist on either of your computers when you pushed. If the changes are incompatible but don't conflict, you may get issues that are difficult to diagnose. This is different than using a Git server — in Git, you can fully test the state on your client system before publishing it, whereas in SVN, you can't ever be certain that the states immediately before commit and after commit are identical.

You should also run this command to pull in changes from the Subversion server, even if you're not ready to commit yourself. You can run `git svn fetch` to grab the new data, but `git svn rebase` does the fetch and then updates your local commits.

```
$ git svn rebase
M    generate_descriptor_proto.sh
r82 = bd16df9173e424c6f52c337ab6efa7f7643282f1 (trunk)
First, rewinding head to replay your work on top of it...
Fast-forwarded master to refs/remotes/trunk.
```

Running `git svn rebase` every once in a while makes sure your code is always up to date. You need to be sure your working directory is clean when you run this, though. If you have local changes, you must either stash your work or temporarily commit it before running `git svn rebase` — otherwise, the command will stop if it sees that the rebase will result in a merge conflict.

## Git Branching Issues

When you've become comfortable with a Git workflow, you'll likely create topic branches, do work on them, and then merge them in. If you're pushing to a Subversion server via `git svn`, you may want to rebase your work onto a single branch each time instead of merging branches together. The reason to prefer rebasing is that Subversion has a linear history and doesn't deal with merges like Git does, so `git svn` follows only the first parent when converting the snapshots into Subversion commits.

Suppose your history looks like the following: you created an `experiment` branch, did two commits, and then merged them back into `master`. When you `dcommit`, you see output like this:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
M    CHANGES.txt
Committed r85
M    CHANGES.txt
```

r85 = 4bfebeec434d156c36f2bcd18f4e3d97dc3269a2 (trunk)

No changes between current HEAD and refs/remotes/trunk

Resetting to the latest refs/remotes/trunk

COPYING.txt: locally modified

INSTALL.txt: locally modified

M COPYING.txt

M INSTALL.txt

Committed r86

M INSTALL.txt

M COPYING.txt

r86 = 2647f6b86ccfcaad4ec58c520e369ec81f7c283c (trunk)

No changes between current HEAD and refs/remotes/trunk

Resetting to the latest refs/remotes/trunk

Running **dcommit** on a branch with merged history works fine, except that when you look at your Git project history, it hasn't rewritten either of the commits you made on the **experiment** branch — instead, all those changes appear in the SVN version of the single merge commit.

When someone else clones that work, all they see is the merge commit with all the work squashed into it; they don't see the commit data about where it came from or when it was committed.

## Subversion Branching

Branching in Subversion isn't the same as branching in Git; if you can avoid using it much, that's probably best. However, you can create and commit to branches in Subversion using git svn.

### **Creating a New SVN Branch**

To create a new branch in Subversion, you run **git svn branch [branchname]**:

```
$ git svn branch opera
```

```
Copying file:///tmp/test-svn/trunk at r87 to file:///tmp/test-svn/branches/opera...
```

```
Found possible branch point: file:///tmp/test-svn/trunk => \
```

```
file:///tmp/test-svn/branches/opera, 87
```

```
Found branch parent: (opera) 1f6bfe471083cbca06ac8d4176f7ad4de0d62e5f
```

```
Following parent with do_switch
```

```
Successfully followed parent
```

```
r89 = 9b6fe0b90c5c9adf9165f700897518dbc54a7cbf (opera)
```

This does the equivalent of the `svn copy trunk branches/opera` command in Subversion and operates on the Subversion server. It's important to note that it doesn't check you out into that branch; if you commit at this point, that commit will go to `trunk` on the server, not `opera`.

## Switching Active Branches

Git figures out what branch your dcommits go to by looking for the tip of any of your Subversion branches in your history — you should have only one, and it should be the last one with a `git-svn-id` in your current branch history.

If you want to work on more than one branch simultaneously, you can set up local branches to `dcommit` to specific Subversion branches by starting them at the imported Subversion commit for that branch. If you want an `opera` branch that you can work on separately, you can run

```
$ git branch opera remotes/opera
```

Now, if you want to merge your `opera` branch into `trunk` (your `master` branch), you can do so with a normal `git merge`. But you need to provide a descriptive commit message (via `-m`), or the merge will say "Merge branch opera" instead of something useful.

Remember that although you're using `git merge` to do this operation, and the merge likely will be much easier than it would be in Subversion (because Git will automatically detect the appropriate merge base for you), this isn't a normal Git merge commit. You have to push this data back to a Subversion server that can't handle a commit that tracks more than one parent; so, after you push it up, it will look like a single commit that squashed in all the work of another branch under a single commit. After you merge one branch into another, you can't easily go back and continue working on that branch, as you normally can in Git. The `dcommit` command that you run erases any information that says what branch was merged in, so subsequent merge-base calculations will be wrong — the `dcommit` makes your `git merge` result look like you ran `git merge --squash`. Unfortunately, there's no good way to avoid this situation — Subversion can't store this information, so you'll always be crippled by its limitations while you're using it as your server. To avoid issues, you should delete the local branch (in this case, `opera`) after you merge it into trunk.

## Subversion Commands

The `git svn` toolset provides a number of commands to help ease the transition to Git by providing some functionality that's similar to what you had in Subversion. Here are a few commands that give you what Subversion used to.

## SVN Style History

If you're used to Subversion and want to see your history in SVN output style, you can run `git svn log` to view your commit history in SVN formatting:

```
$ git svn log
```

```
-----  
r87 | schacon | 2009-05-02 16:07:37 -0700 (Sat, 02 May 2009) | 2 lines
```

```
autogen change
```

```
-----  
r86 | schacon | 2009-05-02 16:00:21 -0700 (Sat, 02 May 2009) | 2 lines
```

```
Merge branch 'experiment'
```

```
-----  
r85 | schacon | 2009-05-02 16:00:09 -0700 (Sat, 02 May 2009) | 2 lines
```

```
updated the changelog
```

You should know two important things about `git svn log`. First, it works offline, unlike the real `svn log` command, which asks the Subversion server for the data. Second, it only shows you commits that have been committed up to the Subversion server. Local Git commits that you haven't committed don't show up; neither do commits that people have made to the Subversion server in the meantime. It's more like the last known state of the commits on the Subversion server.

## SVN Annotation

Much as the `git svn log` command simulates the `svn log` command offline, you can get the equivalent of `svn annotate` by running `git svn blame [FILE]`. The output looks like this:

```
$ git svn blame README.txt
```

```
2  temporal Protocol Buffers - Google's data interchange format  
2  temporal Copyright 2008 Google Inc.  
2  temporal http://code.google.com/apis/protocolbuffers/  
2  temporal  
22 temporal C++ Installation - Unix  
22 temporal =====  
2  temporal  
79  schacon Committing in git-svn.
```

78 schacon

2 temporal To build and install the C++ Protocol Buffer runtime and the Protocol

2 temporal Buffer compiler (protoc) execute the following:

2 temporal

Again, it doesn't show commits that you did locally in Git or that have been pushed to Subversion in the meantime.

### SVN Server Information

You can also get the same sort of information that `svn info` gives you by running `git svn info`:

```
$ git svn info
```

```
Path: .
```

```
URL: https://schacon-test.googlecode.com/svn/trunk
```

```
Repository Root: https://schacon-test.googlecode.com/svn
```

```
Repository UUID: 4c93b258-373f-11de-be05-5f7a86268029
```

```
Revision: 87
```

```
Node Kind: directory
```

```
Schedule: normal
```

```
Last Changed Author: schacon
```

```
Last Changed Rev: 87
```

```
Last Changed Date: 2009-05-02 16:07:37 -0700 (Sat, 02 May 2009)
```

This is like `blame` and `log` in that it runs offline and is up to date only as of the last time you communicated with the Subversion server.

### Ignoring What Subversion Ignores

If you clone a Subversion repository that has `svn:ignore` properties set anywhere, you'll likely want to set corresponding `.gitignore` files so you don't accidentally commit files that you shouldn't. `git svn` has two commands to help with this issue. The first is `git svn create-ignore`, which automatically creates corresponding `.gitignore` files for you so your next commit can include them.

The second command is `git svn show-ignore`, which prints to stdout the lines you need to put in a `.gitignore` file so you can redirect the output into your project exclude file:

```
$ git svn show-ignore > .git/info/exclude
```

That way, you don't litter the project with `.gitignore` files. This is a good option if you're the only Git user on a Subversion team, and your teammates don't want `.gitignore` files in the project.

### Git-Svn Summary

The `git svn` tools are useful if you're stuck with a Subversion server for now or are otherwise in a development environment that necessitates running a Subversion server. You should consider it crippled Git, however, or you'll hit issues in translation that may confuse you and your collaborators. To stay out of trouble, try to follow these guidelines:

- Keep a linear Git history that doesn't contain merge commits made by `git merge`. Rebase any work you do outside of your mainline branch back onto it; don't merge it in.
- Don't set up and collaborate on a separate Git server. Possibly have one to speed up clones for new developers, but don't push anything to it that doesn't have a `git-svn-id` entry. You may even want to add a `pre-receive` hook that checks each commit message for a `git-svn-id` and rejects pushes that contain commits without it.

If you follow those guidelines, working with a Subversion server can be more bearable. However, if it's possible to move to a real Git server, doing so can gain your team a lot more.

## 8.2 Git y Otros Sistemas - Migrating to Git

### **Migrating to Git**

If you have an existing codebase in another VCS but you've decided to start using Git, you must migrate your project one way or another. This section goes over some importers that are included with Git for common systems and then demonstrates how to develop your own custom importer.

### Importing

You'll learn how to import data from two of the bigger professionally used SCM systems — Subversion and Perforce — both because they make up the majority of users I hear of who are currently switching, and because high-quality tools for both systems are distributed with Git.

### Subversion

If you read the previous section about using `git svn`, you can easily use those instructions to `git svn clone` a repository; then, stop using the Subversion server, push to a new Git server, and start using that. If you want the history, you can accomplish that as quickly as you can pull the data out of the Subversion server (which may take a while).

However, the import isn't perfect; and because it will take so long, you may as well do it right. The first problem is the author information. In Subversion, each person committing has a user on the system who is recorded in the commit information. The examples in the previous section show `schacon` in some places, such as the `blame` output and the `git svn log`. If you want to map this to better Git author data, you need a mapping from the Subversion users to the Git authors. Create a file called `users.txt` that has this mapping in a format like this:

```
schacon = Scott Chacon <schacon@geemail.com>
selse = Someo Nelse <selse@geemail.com>
```

To get a list of the author names that SVN uses, you can run this:

```
$ svn log ^/ --xml | grep -P "^<author" | sort -u | \
  perl -pe 's/<author>(.*?)</author>/$1 = /' > users.txt
```

That gives you the log output in XML format — you can look for the authors, create a unique list, and then strip out the XML. (Obviously this only works on a machine with `grep`, `sort`, and `perl` installed.) Then, redirect that output into your `users.txt` file so you can add the equivalent Git user data next to each entry.

You can provide this file to `git svn` to help it map the author data more accurately. You can also tell `git svn` not to include the metadata that Subversion normally imports, by passing `--no-metadata` to the `clone` or `init` command. This makes your `import` command look like this:

```
$ git-svn clone http://my-project.googlecode.com/svn/ \
  --authors-file=users.txt --no-metadata -s my_project
```

Now you should have a nicer Subversion import in your `my_project` directory. Instead of commits that look like this

```
commit 37efa680e8473b615de980fa935944215428a35a
Author: schacon <schacon@4c93b258-373f-11de-be05-5f7a86268029>
Date: Sun May 3 00:12:22 2009 +0000
```

fixed install - go to trunk

```
git-svn-id: https://my-project.googlecode.com/svn/trunk@94
4c93b258-373f-11de-
be05-5f7a86268029
```

they look like this:

```
commit 03a8785f44c8ea5cdb0e8834b7c8e6c469be2ff2
Author: Scott Chacon <schacon@geemail.com>
Date: Sun May 3 00:12:22 2009 +0000
```

fixed install - go to trunk

Not only does the Author field look a lot better, but the **git-svn-id** is no longer there, either.

You need to do a bit of **post-import** cleanup. For one thing, you should clean up the weird references that **git svn** set up. First you'll move the tags so they're actual tags rather than strange remote branches, and then you'll move the rest of the branches so they're local.

To move the tags to be proper Git tags, run

```
$ cp -Rf .git/refs/remotes/tags/* .git/refs/tags/
$ rm -Rf .git/refs/remotes/tags
```

This takes the references that were remote branches that started with **tag/** and makes them real (lightweight) tags.

Next, move the rest of the references under **refs/remotes** to be local branches:

```
$ cp -Rf .git/refs/remotes/* .git/refs/heads/
$ rm -Rf .git/refs/remotes
```

Now all the old branches are real Git branches and all the old tags are real Git tags. The last thing to do is add your new Git server as a remote and push to it. Because you want all your branches and tags to go up, you can run this:

```
$ git push origin --all
```



All your branches and tags should be on your new Git server in a nice, clean import.

## Perforce

The next system you'll look at importing from is Perforce. A Perforce importer is also distributed with Git, but only in the `contrib` section of the source code — it isn't available by default like `git svn`. To run it, you must get the Git source code, which you can download from `git.kernel.org`:

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
$ cd git/contrib/fast-import
```

In this `fast-import` directory, you should find an executable Python script named `git-p4`. You must have Python and the `p4` tool installed on your machine for this import to work. For example, you'll import the Jam project from the Perforce Public Depot. To set up your client, you must export the `P4PORT` environment variable to point to the Perforce depot:

```
$ export P4PORT=public.perforce.com:1666
```

Run the `git-p4 clone` command to import the Jam project from the Perforce server, supplying the depot and project path and the path into which you want to import the project:

```
$ git-p4 clone //public/jam/src@all /opt/p4import
Importing from //public/jam/src@all into /opt/p4import
Reinitialized existing Git repository in /opt/p4import/.git/
Import destination: refs/remotes/p4/master
Importing revision 4409 (100%)
```

If you go to the `/opt/p4import` directory and run `git log`, you can see your imported work:

```
$ git log -2
commit 1fd4ec126171790efd2db83548b85b1bbbc07dc2
Author: Perforce staff <support@perforce.com>
Date: Thu Aug 19 10:18:45 2004 -0800
```

Drop 'rc3' moniker of jam-2.5. Folded rc2 and rc3 RELNOTES into the main part of the document. Built new tar/zip balls.

Only 16 months later.

```
[git-p4: depot-paths = "//public/jam/src/": change = 4409]
```

```
commit ca8870db541a23ed867f38847eda65bf4363371d
Author: Richard Geiger <rmg@perforce.com>
Date: Tue Apr 22 20:51:34 2003 -0800
```

```
Update derived jamgram.c
```

```
[git-p4: depot-paths = "//public/jam/src/": change = 3108]
```

You can see the **git-p4** identifier in each commit. It's fine to keep that identifier there, in case you need to reference the Perforce change number later. However, if you'd like to remove the identifier, now is the time to do so — before you start doing work on the new repository. You can use **git filter-branch** to remove the identifier strings en masse:

```
$ git filter-branch --msg-filter '
    sed -e "/^\[git-p4:/d"
```

```
Rewrite 1fd4ec126171790efd2db83548b85b1bbbc07dc2 (123/123)
Ref 'refs/heads/master' was rewritten
```

If you run **git log**, you can see that all the SHA-1 checksums for the commits have changed, but the **git-p4** strings are no longer in the commit messages:

```
$ git log -2
commit 10a16d60cffca14d454a15c6164378f4082bc5b0
Author: Perforce staff <support@perforce.com>
Date: Thu Aug 19 10:18:45 2004 -0800
```

```
Drop 'rc3' moniker of jam-2.5. Folded rc2 and rc3 RELNOTES into
the main part of the document. Built new tar/zip balls.
```

```
Only 16 months later.
```

```
commit 2b6c6db311dd76c34c66ec1c40a49405e6b527b2
Author: Richard Geiger <rmg@perforce.com>
Date: Tue Apr 22 20:51:34 2003 -0800
```

```
Update derived jamgram.c
```

Your import is ready to push up to your new Git server.

## A Custom Importer

If your system isn't Subversion or Perforce, you should look for an importer online — quality importers are available for CVS, Clear Case, Visual Source Safe, even a directory of archives. If none of these tools works for you, you have a rarer tool, or you otherwise need a more custom importing process, you should use `git fast-import`. This command reads simple instructions from stdin to write specific Git data. It's much easier to create Git objects this way than to run the raw Git commands or try to write the raw objects (see Chapter 9 for more information). This way, you can write an import script that reads the necessary information out of the system you're importing from and prints straightforward instructions to stdout. You can then run this program and pipe its output through `git fast-import`.

To quickly demonstrate, you'll write a simple importer. Suppose you work in `current`, you back up your project by occasionally copying the directory into a time-stamped `back_YYYY_MM_DD` backup directory, and you want to import this into Git. Your directory structure looks like this:

```
$ ls /opt/import_from
back_2009_01_02
back_2009_01_04
back_2009_01_14
back_2009_02_03
current
```

In order to import a Git directory, you need to review how Git stores its data. As you may remember, Git is fundamentally a linked list of commit objects that point to a snapshot of content. All you have to do is tell `fast-import` what the content snapshots are, what commit data points to them, and the order they go in. Your strategy will be to go through the snapshots one at a time and create commits with the contents of each directory, linking each commit back to the previous one.

As you did in the "An Example Git Enforced Policy" section of Chapter 7, we'll write this in Ruby, because it's what I generally work with and it tends to be easy to read. You can write this example pretty easily in anything you're familiar with — it just needs to print the appropriate information to stdout. And, if you are running on Windows, this means you'll need to take special care to not introduce carriage returns at the end your lines — `git fast-import` is very particular about just wanting line feeds (LF) not the carriage return line feeds (CRLF) that Windows uses.

To begin, you'll change into the target directory and identify every subdirectory, each of which is a snapshot that you want to import as a commit. You'll change into each subdirectory and print the commands necessary to export it. Your basic main loop looks like this:

```
last_mark = nil

# loop through the directories
Dir.chdir(ARGV[0]) do
  Dir.glob("*").each do |dir|
    next if File.file?(dir)

    # move into the target directory
    Dir.chdir(dir) do
      last_mark = print_export(dir, last_mark)
    end
  end
end
```

You run `print_export` inside each directory, which takes the manifest and mark of the previous snapshot and returns the manifest and mark of this one; that way, you can link them properly. "Mark" is the `fast-import` term for an identifier you give to a commit; as you create commits, you give each one a mark that you can use to link to it from other commits. So, the first thing to do in your `print_export` method is generate a mark from the directory name:

```
mark = convert_dir_to_mark(dir)
```

You'll do this by creating an array of directories and using the index value as the mark, because a mark must be an integer. Your method looks like this:

```
$marks = []
def convert_dir_to_mark(dir)
  if !$marks.include?(dir)
    $marks << dir
  end
  ($marks.index(dir) + 1).to_s
end
```

Now that you have an integer representation of your commit, you need a date for the commit metadata. Because the date is expressed in the name of the directory, you'll parse it out. The next line in your `print_export` file is

```
date = convert_dir_to_date(dir)
```

where `convert_dir_to_date` is defined as

```
def convert_dir_to_date(dir)
  if dir == 'current'
    return Time.now().to_i
  else
    dir = dir.gsub('back_', '')
    (year, month, day) = dir.split('_')
    return Time.local(year, month, day).to_i
  end
end
```

That returns an integer value for the date of each directory. The last piece of meta-information you need for each commit is the committer data, which you hardcode in a global variable:

```
$author = 'Scott Chacon <schacon@example.com>'
```

Now you're ready to begin printing out the commit data for your importer. The initial information states that you're defining a commit object and what branch it's on, followed by the mark you've generated, the committer information and commit message, and then the previous commit, if any. The code looks like this:

```
# print the import information
puts 'commit refs/heads/master'
puts 'mark :' + mark
puts "committer #{ $author } #{ date } -0700"
export_data('imported from ' + dir)
puts 'from :' + last_mark if last_mark
```

You hardcode the time zone (-0700) because doing so is easy. If you're importing from another system, you must specify the time zone as an offset. The commit message must be expressed in a special format:

```
data (size)\n(contents)
```

The format consists of the word data, the size of the data to be read, a newline, and finally the data. Because you need to use the same format to specify the file contents later, you create a helper method, `export_data`:

```
def export_data(string)
  print "data #{string.size}\n#{string}"
end
```

All that's left is to specify the file contents for each snapshot. This is easy, because you have each one in a directory — you can print out the `deleteall` command followed by the contents of each file in the directory. Git will then record each snapshot appropriately:

```
puts 'deleteall'
Dir.glob("**/*").each do |file|
  next if !File.file?(file)
  inline_data(file)
end
```

Note: Because many systems think of their revisions as changes from one commit to another, fast-import can also take commands with each commit to specify which files have been added, removed, or modified and what the new contents are. You could calculate the differences between snapshots and provide only this data, but doing so is more complex — you may as well give Git all the data and let it figure it out. If this is better suited to your data, check the `fast-import` man page for details about how to provide your data in this manner.

The format for listing the new file contents or specifying a modified file with the new contents is as follows:

```
M 644 inline path/to/file
data (size)
(file contents)
```

Here, 644 is the mode (if you have executable files, you need to detect and specify 755 instead), and inline says you'll list the contents immediately after this line. Your `inline_data` method looks like this:

```

def inline_data(file, code = 'M', mode = '644')
  content = File.read(file)
  puts "#{code} #{mode} inline #{file}"
  export_data(content)
end

```

You reuse the `export_data` method you defined earlier, because it's the same as the way you specified your commit message data.

The last thing you need to do is to return the current mark so it can be passed to the next iteration:

```

return mark

```

NOTE: If you are running on Windows you'll need to make sure that you add one extra step. As mentioned before, Windows uses CRLF for new line characters while git fast-import expects only LF. To get around this problem and make git fast-import happy, you need to tell ruby to use LF instead of CRLF:

```

$stdout.binmode

```

That's it. If you run this script, you'll get content that looks something like this:

```

$ ruby import.rb /opt/import_from
commit refs/heads/master
mark :1
committer Scott Chacon <schacon@geemail.com> 1230883200 -0700
data 29
imported from back_2009_01_02deleteall
M 644 inline file.rb
data 12
version two
commit refs/heads/master
mark :2
committer Scott Chacon <schacon@geemail.com> 1231056000 -0700
data 29
imported from back_2009_01_04from :1
deleteall
M 644 inline file.rb
data 14

```

```
version three
M 644 inline new.rb
data 16
new version one
(...)
```

To run the importer, pipe this output through **git fast-import** while in the Git directory you want to import into. You can create a new directory and then run **git init** in it for a starting point, and then run your script:

```
$ git init
Initialized empty Git repository in /opt/import_to/.git/
$ ruby import.rb /opt/import_from | git fast-import
git-fast-import statistics:
```

```
-----
Alloc'd objects:      5000
Total objects:        18 (      1 duplicates      )
  blobs  :            7 (      1 duplicates      0 deltas)
  trees  :            6 (      0 duplicates      1 deltas)
  commits:            5 (      0 duplicates      0 deltas)
  tags   :            0 (      0 duplicates      0 deltas)
Total branches:        1 (      1 loads      )
  marks:       1024 (      5 unique      )
  atoms:           3
Memory total:         2255 KiB
  pools:          2098 KiB
  objects:         156 KiB
-----
```

```
-----
pack_report: getpagesize()      =    4096
pack_report: core.packedGitWindowSize =  33554432
pack_report: core.packedGitLimit  = 268435456
pack_report: pack_used_ctr       =      9
pack_report: pack_mmap_calls     =      5
pack_report: pack_open_windows  =      1 /      1
pack_report: pack_mapped        =    1356 /    1356
-----
```



As you can see, when it completes successfully, it gives you a bunch of statistics about what it accomplished. In this case, you imported 18 objects total for 5 commits into 1 branch. Now, you can run `git log` to see your new history:

```
$ git log -2
commit 10bfe7d22ce15ee25b60a824c8982157ca593d41
Author: Scott Chacon <schacon@example.com>
Date:   Sun May 3 12:57:39 2009 -0700
```

imported from current

```
commit 7e519590de754d079dd73b44d695a42c9d2df452
Author: Scott Chacon <schacon@example.com>
Date:   Tue Feb 3 01:00:00 2009 -0700
```

imported from back\_2009\_02\_03

There you go — a nice, clean Git repository. It's important to note that nothing is checked out — you don't have any files in your working directory at first. To get them, you must reset your branch to where `master` is now:

```
$ ls
$ git reset --hard master
HEAD is now at 10bfe7d imported from current
$ ls
file.rb lib
```

You can do a lot more with the `fast-import` tool — handle different modes, binary data, multiple branches and merging, tags, progress indicators, and more. A number of examples of more complex scenarios are available in the `contrib/fast-import` directory of the Git source code; one of the better ones is the `git-p4` script I just covered.

# Chapter 9

## Los entresijos internos de Git

Puedes que hayas llegado a este capítulo saltando desde alguno previo o puede que hayas llegado tras leer todo el resto del libro. --En uno u otro caso, aquí es donde aprenderás acerca del funcionamiento interno y la implementación de Git--. Me parece que esta información es realmente importante para entender cuán útil y potente es Git. Pero algunas personas opinan que puede ser confuso e innecesariamente complejo para novatos. Por ello, lo he puesto al final del libro; de tal forma que puedas leerlo antes o después, en cualquier momento, a lo largo de tu proceso de aprendizaje. Lo dejo en tus manos.

Y, ahora que estamos aquí, comencemos con el tema. Ante todo, por si no estuviera suficientemente claro ya, Git es fundamentalmente un sistema de archivo (filesystem) con un interface de usuario (VCS) escrito sobre él. En breve lo veremos con más detalle.

En los primeros tiempos de Git (principalmente antes de la versión 1.5), el interface de usuario era mucho más complejo, ya que se centraba en el sistema de archivos en lugar de en el VCS. En los últimos años, el IU se ha refinado hasta llegar a ser tan limpio y sencillo de usar como el de cualquier otro sistema; pero frecuentemente, el estereotipo sigue mostrando a Git como complejo y difícil de aprender.

La capa del sistema de archivos que almacena el contenido es increíblemente interesante; por ello, es lo primero que voy a desarrollar en este capítulo. A continuación mostraré los mecanismos de transporte y las tareas de mantenimiento del repositorio que posiblemente necesites usar alguna vez.

### 9.1 Los entresijos internos de Git - Fontanería y porcelana

#### Fontanería y porcelana

Este libro habla acerca de como utilizar Git con más o menos 30 verbos, tales como 'checkout', 'branch', 'remote', etc. Pero, debido al origen de Git como una caja de herramientas para un VCS en lugar de como un completo y amigable sistema VCS, existen unos cuantos verbos para realizar

tareas de bajo nivel y que se diseñaron para poder ser utilizados de forma encadenada al estilo UNIX o para ser utilizados en scripts. Estos comandos son conocidos como los "comandos de fontanería", mientras que los comandos más amigables son conocidos como los "comandos de porcelana".

Los primeros ocho capítulos de este libro se encargan casi exclusivamente de los comandos de porcelana. Pero en este capítulo trataremos sobre todo con los comandos de fontanería; comandos que te darán acceso a los entresijos internos de Git y que te ayudarán a comprender cómo y por qué hace Git lo que hace como lo hace. Estos comandos no están pensados para ser utilizados manualmente desde la línea de comandos; sino más bien para ser utilizados como bloques de construcción para nuevas herramientas y scripts de usuario personalizados.

Cuando lanzas 'git init' sobre una carpeta nueva o sobre una ya existente, Git crea la carpeta auxiliar '.git'; la carpeta donde se ubica prácticamente todo lo almacenado y manipulado por Git. Si deseas hacer una copia de seguridad de tu repositorio, con tan solo copiar esta carpeta a cualquier otro lugar ya tienes tu copia completa. Todo este capítulo se encarga de repasar el contenido en dicha carpeta. Tiene un aspecto como este:

```
$ ls  
HEAD  
branches/  
config  
description  
hooks/  
index  
info/  
objects/  
refs/
```

Puede que veas algunos otros archivos en tu carpeta '.git', pero este es el contenido de un repositorio recién creado tras ejecutar 'git init', --es la estructura por defecto--. La carpeta 'branches' no se utiliza en las últimas versiones de Git, y el archivo 'description' se utiliza solo en el programa GitWeb; por lo que no necesitas preocuparte por ellos. El archivo 'config' contiene las opciones de configuración específicas de este proyecto concreto, y la carpeta 'info' guarda un archivo global de exclusión con los patrones a ignorar además de los presentes en el archivo .gitignore. La carpeta 'hooks' contiene tus scripts, tanto de la parte cliente como de la parte servidor, tal y como se ha visto a detalle en el capítulo 6.

Esto nos deja con cuatro entradas importantes: los archivos 'HEAD' e 'index' y las carpetas 'objects' y 'refs'. Estos elementos forman el núcleo de Git. La carpeta 'objects' guarda el contenido de tu base de datos, la carpeta 'refs' guarda los apuntadores a las confirmaciones de cambios (commits), el archivo 'HEAD' apunta a la rama que tengas activa (checked out) en este momento, y el archivo 'index' es donde Git almacena la información sobre tu área de preparación (staging area). Vamos a mirar en detalle cada una de esas secciones, para ver cómo trabaja Git.

## 9.2 Los entresijos internos de Git - Los objetos Git

### Los objetos Git

Git es un sistema de archivo orientado a contenidos. Estupendo. Y eso, ¿qué significa? Pues significa que el núcleo Git es un simple almacén de claves y valores. Cuando insertas cualquier tipo de contenido, él te devuelve una clave que puedes utilizar para recuperar de nuevo dicho contenido en cualquier momento. Para verlo en acción, puedes utilizar el comando de fontanería 'hash-object'. Este comando coge ciertos datos, los guarda en la carpeta '.git/' y te devuelve la clave bajo la cual se han guardado. Para empezar, inicializa un nuevo repositorio Git y comprueba que la carpeta 'objects' está vacía.

```
$ mkdir test
$ cd test
$ git init
Initialized empty Git repository in /tmp/test/.git/
$ find .git/objects
.git/objects
.git/objects/info
.git/objects/pack
$ find .git/objects -type f
$
```

Git ha inicializado la carpeta 'objects', creando en ella las subcarpetas 'pack' e 'info'; pero aún no hay archivos en ellas. Luego, guarda algo de texto en la base de datos de Git:

```
$ echo 'test content' | git hash-object -w --stdin
d670460b4b4aece5915caf5c68d12f560a9fe3e4
```

La opción '-w' indica a 'hash-object' que ha de guardar el objeto; de otro modo, el comando solo te respondería cual sería su clave. La opción '--stdin' indica al comando de leer desde la entrada estandar stdin; si no lo indicas, 'hash-object' espera encontrar la ubicación de un archivo. La salida del comando es una suma de comprobación (checksum hash) de 40 caracteres. Este checksum hash SHA-1 es una suma de comprobación del contenido que estás guardando más una cabecera; cabecera sobre la que trataremos en breve. En estos momentos, ya puedes comprobar la forma en que Git ha guardado tus datos:

```
$ find .git/objects -type f
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

Como puedes ver, hay un archivo en la carpeta 'objects'. En principio, esta es la forma en que guarda Git los contenidos; como un archivo por cada pieza de contenido, nombrado con la suma de comprobación SHA-1 del contenido y su cabecera. La subcarpeta se nombra con los primeros 2 caracteres del SHA, y el archivo con los restantes 38 caracteres.

Puedes volver a recuperar los contenidos usando el comando 'cat-file'. Este comando es algo así como una "navaja suiza" para inspeccionar objetos Git. Pasandole la opción '-p', puedes indicar al comando 'cat-file' que deduzca el tipo de contenido y te lo muestre adecuadamente:

```
$ git cat-file -p d670460b4b4aece5915caf5c68d12f560a9fe3e4
test content
```

Ahora que sabes cómo añadir contenido a Git y cómo recuperarlo de vuelta. Lo puedes hacer también con el propio contenido de los archivos. Por ejemplo, puedes realizar un control simple de versiones sobre un archivo. Para ello, crea un archivo nuevo y guarda su contenido en tu base de datos:

```
$ echo 'version 1' > test.txt
$ git hash-object -w test.txt
83baae61804e65cc73a7201a7252750c76066a30
```

A continuación, escribe algo más de contenido en el archivo y vuélvelo a guardar:

```
$ echo 'version 2' > test.txt
$ git hash-object -w test.txt
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
```

Tu base de datos contendrá las dos nuevas versiones del archivo, así como el primer contenido que habías guardado en ella antes:

```
$ find .git/objects -type f
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

Podrás revertir el archivo a su primera versión:

```
$ git cat-file -p 83baae61804e65cc73a7201a7252750c76066a30 > test.txt
$ cat test.txt
version 1
```

o a su segunda versión

```
$ git cat-file -p 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a > test.txt
$ cat test.txt
version 2
```

Pero no es práctico esto de andar recordando la clave SHA-1 para cada versión de tu archivo; es más, realmente no estás guardando el nombre de tu archivo en el sistema, --solo su contenido--. Este tipo de objeto se denomina un blob (binary large object). Con la orden 'cat-file -t' puedes comprobar el tipo de cualquier objeto almacenado en Git, sin mas que indicar su clave SHA-1':

```
$ git cat-file -t 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
blob
```

## Objetos tipo arbol

El siguiente tipo de objeto a revisar serán los objetos tipo arbol. Estos se encargan de resolver el problema de guardar un nombre de archivo, a la par que guardamos conjuntamente un grupo de archivos. Git guarda contenido de manera similar a un sistema de archivos UNIX, pero de forma algo más simple. Todo el contenido se guarda como objetos arbol (tree) u objetos binarios (blob). Correspondiendo los árboles a las entradas de carpetas; y correspondiendo los binarios, mas o

menos, a los contenidos de los archivos (inodes). Un objeto tipo árbol tiene una o más entradas de tipo árbol. Y cada una de ellas consta de un puntero SHA-1 a un objeto binario (blob) o a un subárbol, más sus correspondientes datos de modo, tipo y nombre de archivo. Por ejemplo, el árbol que hemos utilizado recientemente en el proyecto simplegit, puede resultar algo así como:

```
$ git cat-file -p master^{tree}
100644 blob a906cb2a4a904a152e80877d4088654daad0c859    README
100644 blob 8f94139338f9404f26296befa88755fc2598c289    Rakefile
040000 tree 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0    lib
```

La sentencia `master^{tree}` indica el objeto árbol apuntado por la última confirmación de cambios (commit) en tu rama principal (master). Fíjate en que la carpeta `lib` no es un objeto binario, sino un apuntador a otro objeto tipo árbol.

```
$ git cat-file -p 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0
100644 blob 47c6340d6459e05787f644c2447d2595f5d3a54b    simplegit.rb
```

Conceptualmente, la información almacenada por Git es algo similar a la Figura 9-1.

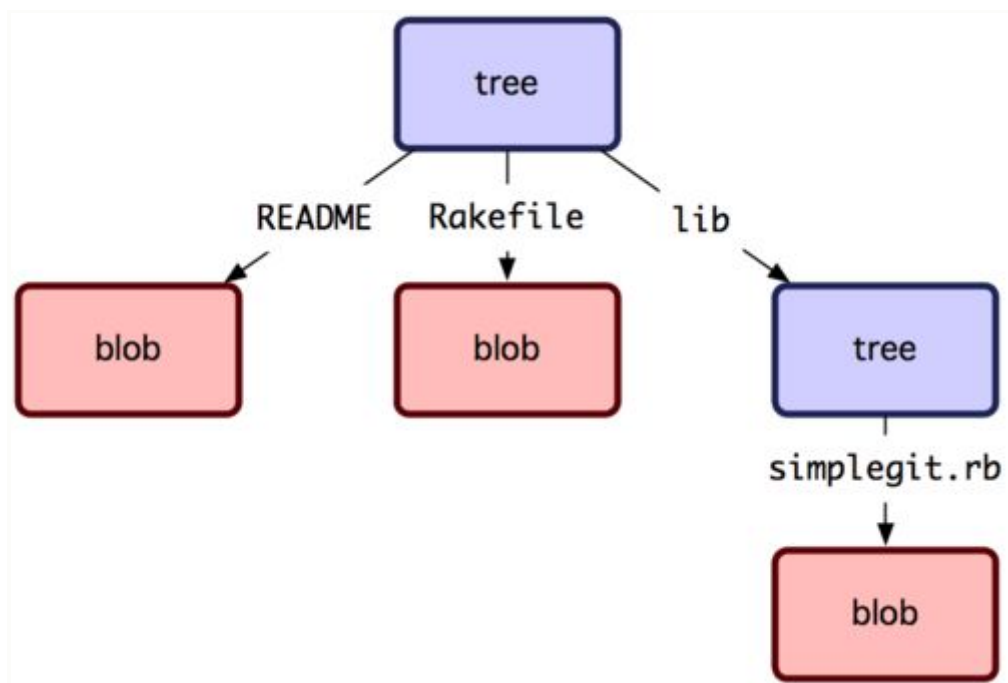


Figura 9-1. Versión simplificada del modelo de datos de Git.

Puedes crear tu propio árbol. Habitualmente Git suele crear un árbol a partir del estado de tu área de preparación (staging area) o índice, escribiendo un objeto árbol con él. Por tanto, para crear un objeto árbol, previamente has de crear un índice preparando algunos archivos para ser almacenados. Puedes utilizar el comando de "fontanería" `update-index` para crear un índice con una sola entrada, --la primera versión de tu archivo `test.txt`--. Este comando se utiliza para añadir artificialmente la versión anterior del archivo `test.txt` a una nueva área de preparación. Has de utilizar la opción `--add`, porque el archivo no existe aún en tu área de preparación (es más, ni siquiera tienes un área de preparación). Y has de utilizar también la opción `--cacheinfo`, porque el archivo que estás añadiendo no está en tu carpeta, sino en tu base de datos. Para terminar, has de indicar el modo, la clave SHA-1 y el nombre de archivo:

```
$ git update-index --add --cacheinfo 100644 \  
83baae61804e65cc73a7201a7252750c76066a30 test.txt
```

En este caso, indicas un modo `100644`, el modo que denota un archivo normal. Otras opciones son `100755`, para un archivo ejecutable; o `120000`, para un enlace simbólico. Estos modos son como los modos de UNIX, pero mucho menos flexibles. Solo estos tres modos son válidos para archivos (blobs) en Git; (aunque también se permiten otros modos para carpetas y submódulos).

Tras esto, puedes usar el comando `write-tree` para escribir el área de preparación a un objeto tipo árbol. Sin necesidad de la opción `-w`, solo llamando al comando `write-tree`, y si dicho árbol no existiera ya, se crea automáticamente un objeto tipo árbol a partir del estado del índice.

```
$ git write-tree  
d8329fc1cc938780ffdd9f94e0d364e0ea74f579  
$ git cat-file -p d8329fc1cc938780ffdd9f94e0d364e0ea74f579  
100644 blob 83baae61804e65cc73a7201a7252750c76066a30    test.txt
```

También puedes comprobar si realmente es un objeto tipo árbol:

```
$ git cat-file -t d8329fc1cc938780ffdd9f94e0d364e0ea74f579  
tree
```

Vamos a crear un nuevo árbol con la segunda versión del archivo `test.txt` y con un nuevo archivo.

```
$ echo 'new file' > new.txt  
$ git update-index test.txt
```



```
$ git update-index --add new.txt
```

El área de preparación contendrá ahora la nueva versión de test.txt, así como el nuevo archivo new.txt. Escribiendo este árbol, (guardando el estado del área de preparación o índice), podrás ver que aparece algo así como:

```
$ git write-tree
0155eb4229851634a0f03eb265b69f5a2d56f341
$ git cat-file -p 0155eb4229851634a0f03eb265b69f5a2d56f341
100644 blob fa49b077972391ad58037050f2a75f74e3671e92    new.txt
100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a    test.txt
```

Aquí se ven las entradas para los dos archivos y también el que la suma de comprobación SHA-1 de test.txt es la "segunda versión" de la anterior (**1f7a7a**). Simplemente por diversión, puedes añadir el primer árbol como una subcarpeta de este otro. Para leer árboles al área de preparación puedes utilizar el comando **read-tree**. Y, en este caso, puedes hacerlo como si fuera una subcarpeta utilizando la opción **--prefix**:

```
$ git read-tree --prefix=bak d8329fc1cc938780ffdd9f94e0d364e0ea74f579
$ git write-tree
3c4e9cd789d88d8d89c1073707c3585e41b0e614
$ git cat-file -p 3c4e9cd789d88d8d89c1073707c3585e41b0e614
040000 tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579    bak
100644 blob fa49b077972391ad58037050f2a75f74e3671e92    new.txt
100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a    test.txt
```

Si crearas una carpeta de trabajo a partir de este nuevo árbol que acabas de escribir, obtendrías los dos archivos en el nivel principal de la carpeta de trabajo y una subcarpeta llamada **bak** conteniendo la primera versión del archivo test.txt. Puedes pensar en algo parecido a la Figura 9-2 para representar los datos guardados por Git para estas estructuras.

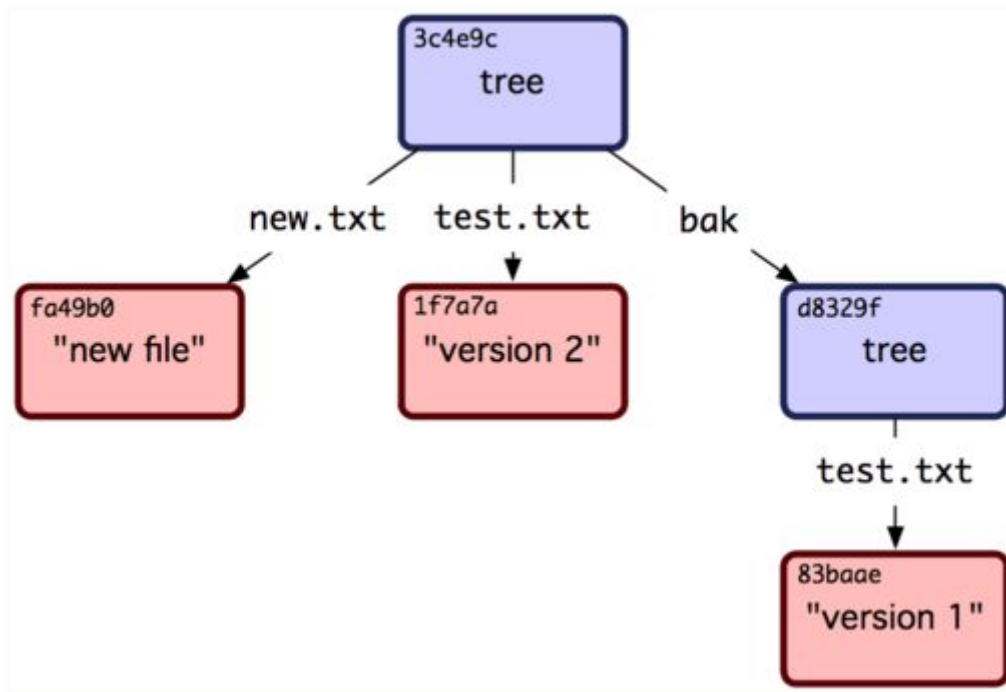


Figura 9-2. La estructura del contenido Git para tus datos actuales.

### Objetos de confirmación de cambios

Tienes tres árboles que representan diferentes momentos interesantes de tu proyecto, pero el problema principal sigue siendo el estar obligado a recordar los tres valores SHA-1 para poder recuperar cualquiera de esos momentos. Asimismo, careces de información alguna sobre quién guardó las instantáneas de esos momentos, cuándo fueron guardados o por qué se guardaron. Este es el tipo de información que almacenan para tí los objetos de confirmación de cambios.

Para crearlos, tan solo has de llamar al comando **commit-tree**, indicando uno de los árboles SHA-1 y los objetos de confirmación de cambios que lo preceden (si es que lo precede alguno). Empezando por el primer árbol que has escrito:

```
$ echo 'first commit' | git commit-tree d8329f
fdf4fc3344e67ab068f836878b6c4951e3b15f3d
```

Con el comando **cat-file** puedes revisar el nuevo objeto de confirmación de cambios recién creado:

```
$ git cat-file -p fdf4fc3
tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579
author Scott Chacon <schacon@gmail.com> 1243040974 -0700
```

committer Scott Chacon <schacon@gmail.com> 1243040974 -0700

### first commit

El formato para un objeto de confirmación de cambios (commit) es sencillo, contemplando: el objeto tipo árbol para la situación del proyecto en ese momento puntual; la información sobre el autor/confirmador, recogida desde las opciones de configuración `user.name` y `user.email`; la fecha y hora actuales; una línea en blanco; y el mensaje de la confirmación de cambios.

Puedes seguir adelante, escribiendo los otros dos objetos de confirmación de cambios. Y relacionando cada uno de ellos con su inmediato anterior:

```
$ echo 'second commit' | git commit-tree 0155eb -p fdf4fc3
cac0cab538b970a37ea1e769cbbde608743bc96d
$ echo 'third commit' | git commit-tree 3c4e9c -p cac0cab
1a410efbd13591db07496601ebc7a059dd55cfe9
```

Cada uno de estos tres objetos de confirmación de cambios apunta a uno de los tres objetos tipo árbol que habías creado previamente. Más aún, en estos momentos tienes ya un verdadero historial Git. Lo puedes comprobar con el comando `git log`. Lanzándolo mientras estás en la última de las confirmaciones de cambio.

```
$ git log --stat 1a410e commit 1a410efbd13591db07496601ebc7a059dd55cfe9
Author: Scott Chacon <schacon@gmail.com>
Date: Fri May 22 18:15:24 2009 -0700
```

### third commit

```
bak/test.txt | 1 +
1 files changed, 1 insertions(+), 0 deletions(-)
```

```
commit cac0cab538b970a37ea1e769cbbde608743bc96d
Author: Scott Chacon <schacon@gmail.com>
Date: Fri May 22 18:14:29 2009 -0700
```

### second commit

```
new.txt | 1 +
test.txt | 2 +-
2 files changed, 2 insertions(+), 1 deletions(-)
```

```
commit fdf4fc3344e67ab068f836878b6c4951e3b15f3d
Author: Scott Chacon <schacon@gmail.com>
Date: Fri May 22 18:09:34 2009 -0700
```

first commit

```
test.txt | 1 +
1 files changed, 1 insertions(+), 0 deletions(-)
```

¡Sorprendente!. Acabas de confeccionar un historial Git utilizando solamente operaciones de bajo nivel, sin usar ninguno de los interfaces principales. Esto es básicamente lo que hace Git cada vez que utilizas los comandos **git add** y **git commit**: guardar objetos binarios (blobs) para los archivos modificados, actualizar el índice, escribir árboles (trees), escribir objetos de confirmación de cambios (commits) que los referencian y relacionar cada uno de ellos con su inmediato precedente. Estos tres objetos Git, -binario, árbol y confirmación de cambios--, se guardan como archivos separados en la carpeta **.git/objects**. Aquí se muestran todos los objetos presentes en este momento en la carpeta del ejemplo, con comentarios acerca de lo que almacena cada uno de ellos:

```
$ find .git/objects -type f
.git/objects/01/55eb4229851634a0f03eb265b69f5a2d56f341 # tree 2
.git/objects/1a/410efbd13591db07496601ebc7a059dd55cfe9 # commit 3
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a # test.txt v2
.git/objects/3c/4e9cd789d88d8d89c1073707c3585e41b0e614 # tree 3
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30 # test.txt v1
.git/objects/ca/c0cab538b970a37ea1e769cbbde608743bc96d # commit 2
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4 # 'test content'
.git/objects/d8/329fc1cc938780ffdd9f94e0d364e0ea74f579 # tree 1
.git/objects/fa/49b077972391ad58037050f2a75f74e3671e92 # new.txt
.git/objects/fd/f4fc3344e67ab068f836878b6c4951e3b15f3d # commit 1
```

Siguiendo todos los enlaces internos, puedes llegar a un gráfico similar al de la figura 9-3.

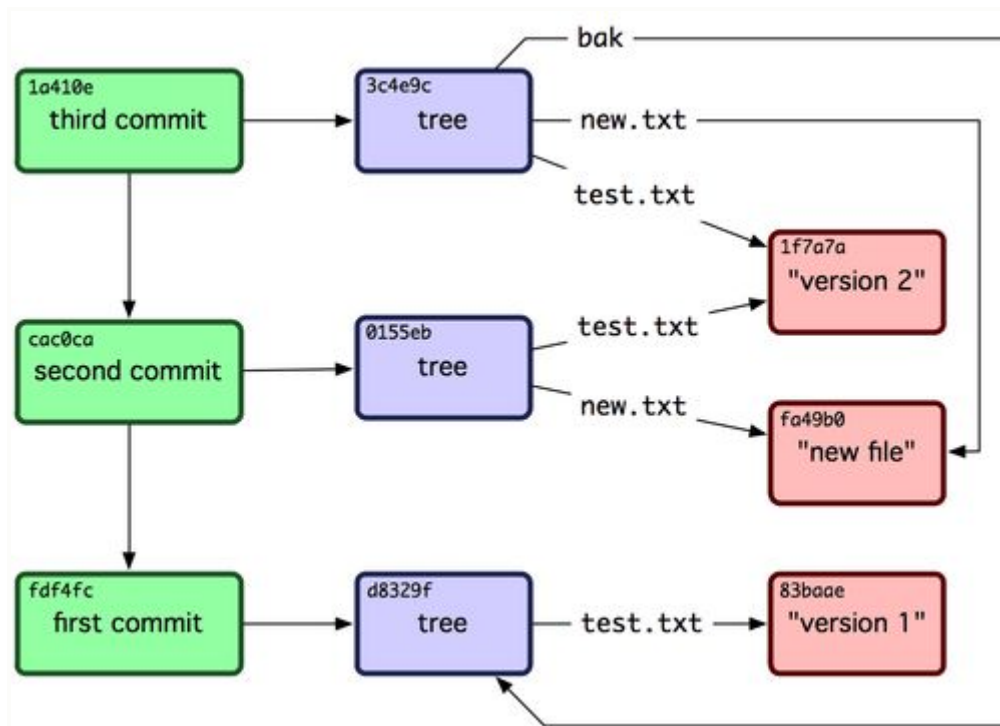


Figura 9-3. Todos los objetos en tu carpeta Git.

## Almacenamiento de los objetos

He citado anteriormente que siempre se almacena una cabecera junto al contenido. Vamos a echar un vistazo a cómo Git almacena sus objetos. Te mostraré el proceso de guardar un objeto binario grande (blob), --en este caso la cadena de texto "what is up, doc?" (¿qué hay de nuevo, viejo?)--, interactivamente, en el lenguaje de script Ruby. Puedes arrancar el modo interactivo de Ruby con el comando **irb**:

```
$ irb
>> content = "what is up, doc?"
=> "what is up, doc?"
```

Git construye la cabecera comenzando por el tipo de objeto, en este caso un objeto binario grande (blob). Después añade un espacio, seguido del tamaño del contenido y termina con un byte nulo:

```
>> header = "blob #{content.length}\0"
=> "blob 16\000"
```

Git concatena la cabecera y el contenido original, para calcular la suma de control SHA-1 conjunta. En Ruby, para calcular el valor SHA-1 de una cadena de texto: has de incluir la librería de generación SHA1 con el comando `require` y llamar luego a la orden

`Digest::SHA1.hexdigest()`:

```
>> store = header + content
=> "blob 16\000what is up, doc?"
>> require 'digest/sha1'
=> true
>> sha1 = Digest::SHA1.hexdigest(store)
=> "bd9dbf5aae1a3862dd1526723246b20206e5fc37"
```

Git comprime todo el contenido con zlib. Y tu puedes hacer lo mismo en Ruby con la librería zlib. Primero has de incluir la librería y luego lanzar la orden `Zlib::Deflate.deflate()` sobre el contenido:

```
>> require 'zlib'
=> true
>> zlib_content = Zlib::Deflate.deflate(store)
=> "\x234K\312\311OR04c(\317H,Q\310,V(-\320QH\311O\266\a\000_\034\a\235"
```

Para terminar, has de escribir el contenido comprimido en un objeto en disco. Para fijar el lugar donde almacenarlo, utilizaremos como nombre de carpeta los dos primeros caracteres del valor SHA-1 y como nombre de archivo los restantes 38 caracteres de dicho valor SHA-1. En Ruby, puedes utilizar la función `FileUtils.mkdir_p()` para crear una carpeta. Después, puedes abrir un archivo con la orden `File.open()` y escribir contenido en él con la orden `write()`:

```
>> path = '.git/objects/' + sha1[0,2] + '/' + sha1[2,38]
=> ".git/objects/bd/9dbf5aae1a3862dd1526723246b20206e5fc37"
>> require 'fileutils'
=> true
>> FileUtils.mkdir_p(File.dirname(path))
=> ".git/objects/bd"
>> File.open(path, 'w') { |f| f.write zlib_content }
=> 32
```

Y uesto es todo!. --acabas de crear un auténtico objeto Git binario grande (blob)--. Todos los demas objetos Git se almacenan de forma similar, pero con la diferencia de que sus cabeceras comienzan con un tipo diferente. En lugar de 'blob' (objeto binario grande), comenzarán por

'commit' (confirmación de cambios), o por 'tree' (árbol). Además, el contenido de un binario (blob) puede ser prácticamente cualquier cosa. Mientras que el contenido de una confirmación de cambios (commit) o de un árbol (tree) han de seguir unos formatos internos muy concretos.

## 9.3 Los entresijos internos de Git - Referencias Git

### Referencias Git

Puedes utilizar algo así como `git log 1a410e` para hechar un vistazo a lo largo de toda tu historia, recorriéndola y encontrando todos tus objetos. Pero para ello has necesitado recordar que la última confirmación de cambios es `1a410e`. Necesitarías un archivo donde almacenar los valores de las sumas de comprobación SHA-1, junto con sus respectivos nombres simples que puedas utilizar como enlaces en lugar de la propia suma de comprobación.

En Git, esto es lo que se conoce como "referencias" o "refs". En la carpeta `.git/refs` puedes encontrar esos archivos con valores SHA-1 y nombres. En el proyecto actual, la carpeta aún no contiene archivos, pero sí contiene una estructura simple:

```
$ find .git/refs
.git/refs
.git/refs/heads
.git/refs/tags
$ find .git/refs -type f
$
```

Para crear una nueva referencia que te sirva de ayuda para recordar cual es tu última confirmación de cambios, puedes realizar técnicamente algo tan simple como:

```
$ echo "1a410efbd13591db07496601ebc7a059dd55cfe9" > .git/refs/heads/master
```

A partir de ese momento, puedes utilizar esa referencia principal que acabas de crear, en lugar del valor SHA-1, en todos tus comandos:

```
$ git log --pretty=oneline master
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
```

cac0cab538b970a37ea1e769cbbde608743bc96d second commit  
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit

No es conveniente editar directamente los archivos de referencia. Git suministra un comando mucho más seguro para hacer esto. Si necesitas actualizar una referencia, puedes utilizar el comando `update-ref`:

```
$ git update-ref refs/heads/master 1a410efbd13591db07496601ebc7a059dd55cfe9
```

Esto es lo que es básicamente una rama en Git: un simple apuntador o referencia a la cabeza de una línea de trabajo. Para crear una rama hacia la segunda confirmación de cambios, puedes hacer:

```
$ git update-ref refs/heads/test cac0ca
```

Y la rama contendrá únicamente trabajo desde esa confirmación de cambios hacia atrás.

```
$ git log --pretty=oneline test  
cac0cab538b970a37ea1e769cbbde608743bc96d second commit  
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

En estos momentos, tu base de datos Git se parecerá conceptualmente a la figura 9-4.

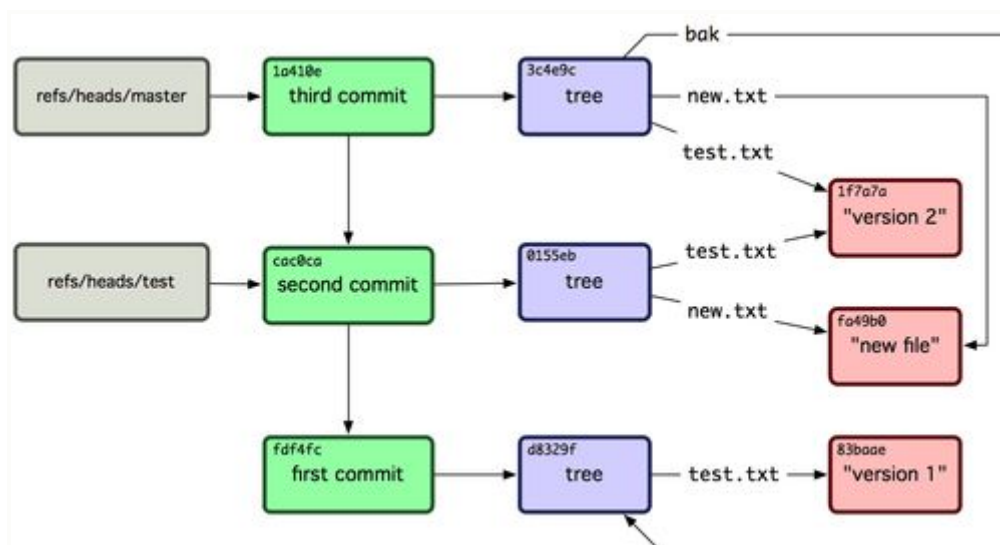


Figura 9-4. Objetos en la carpeta Git, con referencias a las cabeceras de las ramas.



Cuando lanzas comandos como `git branch (nombredelrama)`. Lo que hace Git es añadir, a cualquier nueva referencia que vayas a crear, el valor SHA-1 de la última confirmación de cambios en esa rama.

## La CABEZA (HEAD)

Y ahora nos preguntamos, al lanzar el comando `git branch (nombredelrama)`, ¿cómo sabe Git cuál es el valor SHA-1 de la última confirmación de cambios?. La respuesta a esta pregunta es el archivo HEAD (CABEZA). El archivo HEAD es una referencia simbólica a la rama donde te encuentras en cada momento. Por referencia simbólica me refiero a que, a diferencia de una referencia normal, esta contiene un enlace a otra referencia en lugar de un valor SHA-1. Si miras dentro del archivo, podrás observar algo como:

```
$ cat .git/HEAD
ref: refs/heads/master
```

Si lanzas el comando `git checkout test`, Git actualiza el contenido del archivo:

```
$ cat .git/HEAD
ref: refs/heads/test
```

Cuando lanzas una orden `git commit`, se crea un nuevo objeto de confirmación de cambios teniendo como padre la confirmación con valor SHA-1 a la que en ese momento esté apuntando la referencia en HEAD.

Puedes editar manualmente este archivo. Pero, también para esta tarea existe un comando más seguro: `symbolic-ref`. Puedes leer el valor de HEAD a través de él:

```
$ git symbolic-ref HEAD
refs/heads/master
```

Y también puedes cambiar el valor de HEAD a través de él:

```
$ git symbolic-ref HEAD refs/heads/test
$ cat .git/HEAD
ref: refs/heads/test
```

Pero no puedes fijar una referencia simbólica fuera de "refs":

```
$ git symbolic-ref HEAD test
fatal: Refusing to point HEAD outside of refs/
```

## Etiquetas

Acabas de conocer los tres principales tipos de objetos Git, pero hay un cuarto. El objeto tipo etiqueta es muy parecido al tipo confirmación de cambios, --contiene un marcador, una fecha, un mensaje y un enlace--. Su principal diferencia reside en que apunta a una confirmación de cambios (commit) en lugar de a un árbol (tree). Es como una referencia a una rama, pero permaneciendo siempre inmovil, --apuntando siempre a la misma confirmación de cambios--, dándo un nombre mas amigable a esta.

Tal y como se ha comentado en el capítulo 2, hay dos tipos de etiquetas: las anotativas y las ligeras. Puedes crear una etiqueta ligera lanzando un comando tal como:

```
$ git update-ref refs/tags/v1.0 cac0cab538b970a37ea1e769cbbde608743bc96d
```

Una etiqueta ligera es simplemente eso: una rama que nunca se mueve. Sin embargo, una etiqueta anotativa es más compleja. Al crear una etiqueta anotativa, Git crea un objeto tipo etiqueta y luego escribe una referencia apuntando a él en lugar de apuntar directamente a una confirmación de cambios. Puedes comprobarlo creando una: (la opción **-a** indica que la etiqueta es anotativa)

```
$ git tag -a v1.1 1a410efbd13591db07496601ebc7a059dd55cfe9 -m 'test tag'
```

Este es el objeto SHA-1 creado:

```
$ cat .git/refs/tags/v1.1
9585191f37f7b0fb9444f35a9bf50de191beadc2
```

Ahora, lanzando el comando **cat-file** para ese valor SHA-1:

```
$ git cat-file -p 9585191f37f7b0fb9444f35a9bf50de191beadc2
object 1a410efbd13591db07496601ebc7a059dd55cfe9
type commit
tag v1.1
tagger Scott Chacon <schacon@gmail.com> Sat May 23 16:48:58 2009 -0700
```

test tag

Merece destacar que el inicio del objeto apunta al SHA-1 de la confirmación de cambios recién etiquetada. Y también el que no ha sido necesario apuntar directamente a una confirmación de cambios. Realmente puedes etiquetar cualquier tipo de objeto Git. Por ejemplo, en el código fuente de Git los gestores han añadido su clave GPG pública como un objeto binario (blob) y lo han etiquetado. Puedes ver esta clave pública con el comando

```
$ git cat-file blob junio-gpg-pub
```

lanzado sobre el código fuente de Git. El kernel de Linux tiene también un objeto tipo etiqueta apuntando a un objeto que no es una confirmación de cambios (commit). La primera etiqueta que se creó es la que apunta al árbol (tree) inicial donde se importó el código fuente.

## Remotos

El tercer tipo de referencia que puedes observar es la referencia remota. Si añades un remoto y envías algo a él, Git almacenará en dicho remoto el último valor para cada rama presente en la carpeta **refs/remotes**. Por ejemplo, puedes añadir un remoto denominado **origin** y enviar a él tu rama **master**:

```
$ git remote add origin git@github.com:schacon/simplegit-progit.git
$ git push origin master
Counting objects: 11, done.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (7/7), 716 bytes, done.
Total 7 (delta 2), reused 4 (delta 1)
To git@github.com:schacon/simplegit-progit.git
a11bef0..ca82a6d master -> master
```

Tras lo cual puedes confirmar cual era la rama **master** en el remoto **origin** la última vez que comunicase con el servidor. Comprobando el archivo **refs/remotes/origin/master**:

```
$ cat .git/refs/remotes/origin/master
ca82a6dff817ec66f44342007202690a93763949
```

Las referencias remotas son distintas de las ramas normales, (referencias en **refs/heads**); y no se pueden recuperar (checkout) al espacio de trabajo. Git las utiliza solamente como marcadores al último estado conocido de cada rama en cada servidor remoto declarado.

## 9.4 Los entresijos internos de Git - Archivos empaquetadores

### Archivos empaquetadores

Volviendo a los objetos en la base de datos de tu repositorio Git de pruebas. En este momento, tienes 11 objetos --4 binarios, 3 árboles, 3 confirmaciones de cambios y 1 etiqueta--.

```
$ find .git/objects -type f
.git/objects/01/55eb4229851634a0f03eb265b69f5a2d56f341 # tree 2
.git/objects/1a/410efbd13591db07496601ebc7a059dd55cfe9 # commit 3
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a # test.txt v2
.git/objects/3c/4e9cd789d88d8d89c1073707c3585e41b0e614 # tree 3
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30 # test.txt v1
.git/objects/95/85191f37f7b0fb9444f35a9bf50de191beadc2 # tag
.git/objects/ca/c0cab538b970a37ea1e769cbbde608743bc96d # commit 2
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4 # 'test content'
.git/objects/d8/329fc1cc938780ffdd9f94e0d364e0ea74f579 # tree 1
.git/objects/fa/49b077972391ad58037050f2a75f74e3671e92 # new.txt
.git/objects/fd/f4fc3344e67ab068f836878b6c4951e3b15f3d # commit 1
```

Git comprime todos esos archivos con zlib, por lo que ocupan más bien poco. Entre todos suponen solamente 925 bytes. Puedes añadir algún otro archivo de gran contenido al repositorio. Y verás una interesante funcionalidad de Git. Añadiendo el archivo `repo.rb` de la librería Grit con la que has estado trabajando anteriormente, supondrá añadir un archivo con unos 12 Kbytes de código fuente.

```
$ curl http://github.com/mojombo/grit/raw/master/lib/grit/repo.rb > repo.rb
$ git add repo.rb
$ git commit -m 'added repo.rb'
[master 484a592] added repo.rb
3 files changed, 459 insertions(+), 2 deletions(-)
delete mode 100644 bak/test.txt
create mode 100644 repo.rb
```

```
rewrite test.txt (100%)
```

Si hechas un vistazo al árbol resultante, podrás observar el valor SHA-1 del objeto binario correspondiente a dicho archivo repo.rb:

```
$ git cat-file -p master^{tree}
100644 blob fa49b077972391ad58037050f2a75f74e3671e92    new.txt
100644 blob 9bc1dc421dcd51b4ac296e3e5b6e2a99cf44391e    repo.rb
100644 blob e3f094f522629ae358806b17daf78246c27c007b    test.txt
```

Y ver su tamaño con el comando `git cat-file`:

```
$ git cat-file -s 9bc1dc421dcd51b4ac296e3e5b6e2a99cf44391e
12898
```

Ahora, modifica un poco dicho archivo y comprueba lo que sucede:

```
$ echo '# testing' >> repo.rb
$ git commit -am 'modified repo a bit'
[master ab1afef] modified repo a bit
1 files changed, 1 insertions(+), 0 deletions(-)
```

Revisando el árbol creado por esta última confirmación de cambios, verás algo interesante:

```
$ git cat-file -p master^{tree}
100644 blob fa49b077972391ad58037050f2a75f74e3671e92    new.txt
100644 blob 05408d195263d853f09dca71d55116663690c27c    repo.rb
100644 blob e3f094f522629ae358806b17daf78246c27c007b    test.txt
```

El objeto binario es ahora un binario completamente diferente. Aunque solo has añadido una única línea al final de un archivo que ya contenía 400 líneas, Git ha almacenado el resultado como un objeto completamente nuevo.

```
$ git cat-file -s 05408d195263d853f09dca71d55116663690c27c
12908
```

Y, así, tienes en tu disco dos objetos de 12 Kbytes prácticamente idénticos. ¿No sería práctico si Git pudiera almacenar uno de ellos completo y luego solo las diferencias del segundo con respecto al primero?

Pues bien, Git lo puede hacer. El formato inicial como Git guarda sus objetos en disco es el formato conocido como "relajado" (loose). Pero, sin embargo, de vez en cuando, Git suele agrupar varios de esos objetos en un único binario denominado archivo "empaquetador". Para ahorrar espacio y hacer así más eficiente su almacenamiento. Esto sucede cada vez que tiene demasiados objetos en formato "relajado"; o cuando tu invocas manualmente al comando `git gc`; o justo antes de enviar cualquier cosa a un servidor remoto. Puedes comprobar el proceso pidiéndole expresamente a Git que empaquete objetos, utilizando el comando `git gc`:

```
$ git gc
Counting objects: 17, done.
Delta compression using 2 threads.
Compressing objects: 100% (13/13), done.
Writing objects: 100% (17/17), done.
Total 17 (delta 1), reused 10 (delta 0)
```

Tras esto, si miras los objetos presentes en la carpeta, veras que han desaparecido la mayoría de los que habia anteriormente. Apareciendo un par de objetos nuevos:

```
$ find .git/objects -type f
.git/objects/71/08f7ecb345ee9d0084193f147cdad4d2998293
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
.git/objects/info/packs
.git/objects/pack/pack-7a16e4488ae40c7d2bc56ea2bd43e25212a66c45.idx
.git/objects/pack/pack-7a16e4488ae40c7d2bc56ea2bd43e25212a66c45.pack
```

Solo han quedado aquellos objetos binarios no referenciados por ninguna confirmación de cambios --en este caso, el ejemplo de "¿que hay de nuevo, viejo?" y el ejemplo de "contenido de pruebas"-- Porque nunca los has llegado a incluir en ninguna confirmación de cambios, no se han considerado como objetos definitivos y, por tanto, no han sido empaquetados.

Los otros archivos presentes son el nuevo archivo empaquetador y un índice. El archivo empaquetador es un único archivo conteniendo dentro de él todos los objetos sueltos eliminados del sistema de archivo. El índice es un archivo que contiene las posiciones de cada uno de esos objetos dentro del archivo empaquetador. Permittiendonos así buscarlos y extraer rápidamente cualquiera de ellos. Lo que es interesante es el hecho de que, aunque los objetos originales

presentes en el disco antes del **gc** ocupaban unos 12 Kbytes, el nuevo archivo empaquetador apenas ocupa 6 Kbytes. Empaquetando los objetos, has conseguido reducir a la mitad el uso de disco.

¿Cómo consigue Git esto? Cuando Git empaqueta objetos, va buscando archivos de igual nombre y tamaño similar. Almacenando únicamente las diferencias entre una versión de cada archivo y la siguiente. Puedes comprobarlo mirando en el interior del archivo empaquetador. Y, para eso, has de utilizar el comando "de fontanería" **git verify-pack**:

```
$ git verify-pack -v \
.git/objects/pack/pack-7a16e4488ae40c7d2bc56ea2bd43e25212a66c45.idx
0155eb4229851634a0f03eb265b69f5a2d56f341 tree  71 76 5400
05408d195263d853f09dca71d55116663690c27c blob  12908 3478 874
09f01cea547666f58d6a8d809583841a7c6f0130 tree  106 107 5086
1a410efbd13591db07496601ebc7a059dd55cfe9 commit 225 151 322
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a blob   10 19 5381
3c4e9cd789d88d8d89c1073707c3585e41b0e614 tree  101 105 5211
484a59275031909e19aadb7c92262719cfcdf19a commit 226 153 169
83baae61804e65cc73a7201a7252750c76066a30 blob   10 19 5362
9585191f37f7b0fb9444f35a9bf50de191beadc2 tag    136 127 5476
9bc1dc421dcd51b4ac296e3e5b6e2a99cf44391e blob   7 18 5193 1 \
05408d195263d853f09dca71d55116663690c27c
ab1afef80fac8e34258ff41fc1b867c702daa24b commit 232 157 12
cac0cab538b970a37ea1e769cbbde608743bc96d commit 226 154 473
d8329fc1cc938780ffdd9f94e0d364e0ea74f579 tree   36 46 5316
e3f094f522629ae358806b17daf78246c27c007b blob  1486 734 4352
f8f51d7d8a1760462eca26eebafde32087499533 tree  106 107 749
fa49b077972391ad58037050f2a75f74e3671e92 blob   9 18 856
fdf4fc3344e67ab068f836878b6c4951e3b15f3d commit 177 122 627
chain length = 1: 1 object
pack-7a16e4488ae40c7d2bc56ea2bd43e25212a66c45.pack: ok
```

Puedes observar que el objeto binario **9bc1d**, (correspondiente a la primera versión de tu archivo `repo.rb`), tiene una referencia al binario **05408** (la segunda versión de ese archivo). La tercera columna refleja el tamaño de cada objeto dentro del paquete. Observándose que **05408** ocupa unos 12 Kbytes; pero **9bc1d** solo ocupa 7 bytes. Resulta curioso que se almacene completa la segunda versión del archivo, mientras que la versión original es donde se almacena solo la diferencia. Esto se debe a la mayor probabilidad de que vayamos a recuperar rápidamente la versión más reciente del archivo.

Lo verdaderamente interesante de todo este proceso es que podemos reempaquetar en cualquier momento. De vez en cuando, Git, en su empeño por optimizar la ocupación de espacio, reempaqueta automáticamente toda la base de datos. Pero, también tu mismo puedes reempaquetar en cualquier momento, lanzando manualmente el comando `git gc`.

## 9.5 Los entresijos internos de Git - Las especificaciones para hacer referencia a... (refspec)

### Las especificaciones para hacer referencia a... (refspec)

A lo largo del libro has utilizado sencillos mapeados entre ramas remotas y referencias locales; pero las cosas pueden ser bastante más complejas. Supón que añades un remoto tal que:

```
$ git remote add origin git@github.com:schacon/simplegit-progit.git
```

Esto añade una nueva sección a tu archivo `.git/config`, indicando el nombre del remoto (`origin`), la ubicación (URL) del repositorio remoto y la referencia para recuperar (fetch) desde él:

```
[remote "origin"]
    url = git@github.com:schacon/simplegit-progit.git
    fetch = +refs/heads/*:refs/remotes/origin/*
```

El formato para esta referencia es un signo `+` opcional, seguido de una sentencia `<orig>:<dest>`; donde `<orig>` es la plantilla para referencias en el lado remoto y `<dest>` el lugar donde esas referencias se escribirán en el lado local. El `+`, si está presente, indica a Git que debe actualizar la referencia incluso en los casos en que no se dé un avance rápido (fast-forward).

En el caso por defecto en que es escrito por un comando `git remote add`, Git recupera del servidor todas las referencias bajo `refs/heads/`, y las escribe localmente en `refs/remotes/origin/`. De tal forma que, si existe una rama `master` en el servidor, puedes acceder a ella localmente a través de

```
$ git log origin/master
$ git log remotes/origin/master
```



```
$ git log refs/remotes/origin/master
```

Todas estas sentencias son equivalentes, ya que Git expande cada una de ellas a `refs/remotes/origin/master`.

Si, en cambio, quisieras hacer que Git recupere únicamente la rama `master` y no cualquier otra rama en el servidor remoto. Puedes cambiar la línea de recuperación a

```
fetch = +refs/heads/master:refs/remotes/origin/master
```

Quedando así esta referencia como la referencia por defecto para el comando `git fetch` para ese remoto. Para hacerlo puntualmente en un momento concreto, puedes especificar la referencia directamente en la línea de comando. Para recuperar la rama `master` del servidor remoto a tu rama `origin/mymaster` local, puedes lanzar el comando

```
$ git fetch origin master:refs/remotes/origin/mymaster
```

Puedes incluso indicar múltiples referencias en un solo comando. Escribiendo algo así como:

```
$ git fetch origin master:refs/remotes/origin/mymaster \  
    topic:refs/remotes/origin/topic  
From git@github.com:schacon/simplegit  
! [rejected]      master    -> origin/mymaster (non fast forward)  
* [new branch]   topic     -> origin/topic
```

En este ejemplo, se ha rechazado la recuperación de la rama `master` porque no era una referencia de avance rápido (fast-forward). Puedes forzarlo indicando el signo `+` delante de la referencia.

Es posible asimismo indicar referencias múltiples en el archivo de configuración. Si, por ejemplo, siempre recuperas las ramas `'master'` y `'experiment'`, puedes poner dos líneas:

```
[remote "origin"]  
    url = git@github.com:schacon/simplegit-progit.git  
    fetch = +refs/heads/master:refs/remotes/origin/master  
    fetch = +refs/heads/experiment:refs/remotes/origin/experiment
```

Pero, en ningún caso puedes poner referencias genéricas parciales; por ejemplo, algo como esto sería erróneo:

```
fetch = +refs/heads/qa*:refs/remotes/origin/qa*
```

Aunque, para conseguir algo similar, puedes utilizar los espacios de nombres . Si tienes un equipo QA que envía al servidor una serie de ramas. Y deseas recuperar la rama master y cualquiera otra de las ramas del equipo; pero no recuperar ninguna rama de otro equipo. Puedes utilizar una sección de configuración como esta:

```
[remote "origin"]
    url = git@github.com:schacon/simplegit-progit.git
    fetch = +refs/heads/master:refs/remotes/origin/master
    fetch = +refs/heads/qa/*:refs/remotes/origin/qa/*
```

De esta forma, puedes asignar fácilmente espacios de nombres. Y resolver así complejos flujos de trabajo donde tengas simultáneamente , por ejemplo, un equipo QA enviando ramas, varios desarrolladores enviando ramas también y equipos integradores enviando y colaborando en ramas remotas.

## Enviando (push) referencias

Es útil poder recuperar (fetch) referencias relativas en espacios de nombres, tal y como hemos visto. Pero, ¿cómo pueden enviar (push) sus ramas al espacio de nombres **qa/** los miembros de equipo QA ? . Pues utilizando las referencias (refspecs) para enviar.

Si alguien del equipo QA quiere enviar su rama **master** a la ubicación **qa/master** en el servidor remoto, puede lanzar algo así como:

```
$ git push origin master:refs/heads/qa/master
```

Y, para que se haga de forma automática cada vez que ejecute **git push origin**, puede añadir una entrada **push** a su archivo de configuración:

```
[remote "origin"]
    url = git@github.com:schacon/simplegit-progit.git
    fetch = +refs/heads/*:refs/remotes/origin/*
    push = refs/heads/master:refs/heads/qa/master
```

Esto hará que un simple comando **git push origin** envíe por defecto la rama local **master** a la rama remota **qa/master**,

## Borrando referencias

Se pueden utilizar las referencias (refspec) para borrar en el servidor remoto. Por ejemplo, lanzando algo como:

```
$ git push origin :topic
```

Se elimina la rama 'topic' del servidor remoto, ya que la sustituimos por nada. (Al ser la referencia `<origen>:<destino>`, si no indicamos la parte `<origen>`, realmente estamos diciendo que enviamos 'nada' a `<destino>`.)

# 9.6 Los entresijos internos de Git - Protocolos de transferencia

## Protocolos de transferencia

Git puede transferir datos entre dos repositorios utilizando uno de sus dos principales mecanismos de transporte: sobre HTTP (protocolo tonto), o sobre los denominados protocolos inteligentes (utilizados en `file://`, `ssh://` o `git://`). En esta parte, se verán sucintamente cómo trabajan esos dos tipos de protocolo.

### El protocolo tonto (dumb)

El transporte de Git sobre protocolo HTTP es conocido también como protocolo tonto. Porque no requiere ningún tipo de código Git en la parte servidor. El proceso de recuperación (fetch) de datos se limita a una serie de peticiones GET, siendo el cliente quien ha de conocer la estructura del repositorio Git en el servidor. Vamos a revisar el proceso `http-fetch` para una librería simple de Git:

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

Lo primero que hace este comando es recuperar el archivo `info/refs`. Este es un archivo escrito por el comando `update-server-info`, el que has de habilitar como enganche (hook) `post-receive` para permitir funcionar correctamente al transporte HTTP:

=> GET info/refs

ca82a6dff817ec66f44342007202690a93763949      refs/heads/master

A partir de ahí, ya tienes una lista de las referencias remotas y sus SHAs. Lo siguiente es mirar cual es la referencia a HEAD, de tal forma que puedas saber el punto a activar (checkout) cuando termines:

=> GET HEAD

ref: refs/heads/master

Ves que es la rama **master** la que has de activar cuando el proceso esté completado.

sus ramas al espacio de nombres **qa**/En este punto, ya estás preparado para seguir procesando el resto de los objetos. En el archivo **info/refs** se ve que el punto de partida es la confirmación de cambios (commit) **ca82a6**, y, por tanto, comenzaremos recuperandola:

=> GET objects/ca/82a6dff817ec66f44342007202690a93763949  
(179 bytes of binary data)

Cuando recuperas un objeto, dicho objeto se encuentra suelto (loose) en el servidor y lo traes mediante una petición estática HTTP GET. Puedes descomprimirlo, quitarle la cabecera y mirar el contenido:

```
$ git cat-file -p ca82a6dff817ec66f44342007202690a93763949
tree cfd3bf379e4f8dba8717dee55aab78aef7f4daf
parent 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
author Scott Chacon <schacon@gmail.com> 1205815931 -0700
committer Scott Chacon <schacon@gmail.com> 1240030591 -0700
```

changed the version number

Tras esto, ya tienes más objetos a recuperar --el árbol de contenido **cfd3b** al que apunta la confirmación de cambios; y la confirmación de cambios padre **085bb3**--.

=> GET objects/08/5bb3bcb608e1e8451d4b2432f8ecbe6306e7e7  
(179 bytes of data)

El siguiente objeto confirmación de cambio (commit). Y el árbol de contenido:

=> GET objects/cf/da3bf379e4f8dba8717dee55aab78aef7f4daf  
(404 - Not Found)

Pero... ¡Ay!... parece que el objeto árbol no está suelto en el servidor. Por lo que obtienes una respuesta 404 (objeto no encontrado). Puede haber un par de razones para que suceda esto: el objeto está en otro repositorio alternativo; o el objeto está en este repositorio, pero dentro de un objeto empaquetador (packfile). Git comprueba primero a ver si en el listado hay alguna alternativa:

=> GET objects/info/http-alternates  
(empty file)

En el caso de que esto devolviera una lista de ubicaciones (URL) alternativas, Git busca en ellas. (Es un mecanismo muy adecuado en aquellos proyectos donde hay segmentos derivados uno de otro compartiendo objetos en disco.) Pero, en este caso, no hay alternativas. Por lo que el objeto debe encontrarse dentro de un empaquetado. Para ver que empaquetados hay disponibles en el servidor, has de recuperar el archivo `objects/info/packs`. Este contiene una lista de todos ellos: (que ha sido generada por `update-server-info`)

=> GET objects/info/packs  
P pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack

Vemos que hay un archivo empaquetado, y el objeto buscado ha de encontrarse dentro de él; pero merece comprobarlo revisando el archivo de índice, para asegurarse. Hacer la comprobación es sobre todo útil en aquellos casos donde existan múltiples archivos empaquetados en el servidor, para determinar así en cual de ellos se encuentra el objeto que necesitas:

=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.idx  
(4k of binary data)

Una vez tengas el índice del empaquetado, puedes mirar si el objeto buscado está en él, (Dicho índice contiene la lista de SHAs de los objetos dentro del empaquetado y las ubicaciones -offsets- de cada uno de ellos dentro de él.) Una vez comprobada la presencia del objeto, adelante con la recuperación de todo el archivo empaquetado:

=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack  
(13k of binary data)

Cuando tengas el objeto árbol, puedes continuar avanzando por las confirmaciones de cambio. Y, como estás también están dentro del archivo empaquetado que acabas de descargar, ya no necesitas hacer mas peticiones al servidor. Git activa una copia de trabajo de la rama **master** señalada por la referencia HEAD que has descargado al principio.

La salida completa de todo el proceso es algo como esto:

```
$ git clone http://github.com/schacon/simplegit-progit.git
Initialized empty Git repository in /private/tmp/simplegit-progit/.git/
got ca82a6dff817ec66f44342007202690a93763949
walk ca82a6dff817ec66f44342007202690a93763949
got 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Getting alternates list for http://github.com/schacon/simplegit-progit.git
Getting pack list for http://github.com/schacon/simplegit-progit.git
Getting index for pack 816a9b2334da9953e530f27bcac22082a9f5b835
Getting pack 816a9b2334da9953e530f27bcac22082a9f5b835
  which contains cfda3bf379e4f8dba8717dee55aab78aef7f4daf
walk 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
walk a11bef06a3f659402fe7563abf99ad00de2209e6
```

## El protocolo inteligente (smart)

HTTP es un protocolo simple, pero ineficiente. Es mucho más común utilizar protocolos inteligentes para transferir datos. Estos protocolos suelen tener procesos en el lado remoto y conocen acerca de la estructura de datos Git en ese lado, --pueden leer datos localmente y determinar lo que el cliente tiene ya o necesita a continuación, para generar automáticamente datos expresamente preparados para él--. Existen dos conjuntos de procesos para transferir datos: uno para para enviar y otro par para recibir.

### Enviando datos, (Uploading)

Para enviar datos a un proceso remoto, Git utiliza **send-pack** (enviar paquete) y **receive-pack**(recibir paquete). El proceso **send-pack** corre en el cliente y conecta con el proceso **receive-pack**corriendo en el lado remoto.

Por ejemplo, si lanzas el comando **git push origin master** en tu proyecto y **origin** está definida como una ubicación que utiliza el protocolo SSH. Git lanzará el proceso **send-pack** , con el que

establece conexión SSH con tu servidor. En el servidor remoto, a través de una llamada SSH, intentará lanzar un comando tal como:

```
$ ssh -x git@github.com "git-receive-pack 'schacon/simplegit-progit.git'"
005bca82a6dff817ec66f4437202690a93763949 refs/heads/master report-status
delete-refs
003e085bb3bcb608e1e84b2432f8ecbe6306e7e7 refs/heads/topic
0000
```

El comando `git-receive-pack` responde con una línea por cada una de las referencias que tenga, --en este caso, la rama `master` y su SHA--. La primera línea suele indicar también una lista con las capacidades del servidor, (en este caso `report-status` --dar situación-- y `delete-refs` --borrar referencias--).

Cada línea comienza con un valor de 4 bytes, en hexadecimal, indicando la longitud del resto de la línea. La primera de las líneas comienza con `005b`, 91 en decimal, indicándonos que hay 91 bytes más en esa línea. La siguiente línea comienza con `003e`, 62 en decimal, por lo que has de leer otros 62 bytes hasta el final de la línea. Y la última línea comienza con `0000`, indicando así que la lista de referencias ha terminado.

Con esta información, el proceso `send-pack` ya puede determinar las confirmaciones de cambios (commits) presentes en el servidor. Para cada una de las referencias que se van a actualizar, el proceso `send-pack` llama al proceso `receive-pack` con la información pertinente. Por ejemplo, si estás actualizando la rama `master` y añadiendo otra rama `experiment`, la respuesta del proceso `send-pack` será algo así como:

```
0085ca82a6dff817ec66f44342007202690a93763949
15027957951b64cf874c3557a0f3547bd83b3ff6 refs/heads/master report-status
0067000000000000000000000000000000000000000000000000000000000000
cdfdb42577e2506715f8cfeacdbabc092bf63e8d refs/heads/experiment
0000
```

Una clave SHA-1 con todo '0's, nos indica que no había nada anteriormente, y que, por tanto, estamos añadiendo una nueva referencia. Si estuvieras borrando una referencia existente, verías lo contrario: una clave todo '0's en el lado derecho.

Git envía una línea por cada referencia a actualizar, indicando el viejo SHA, el nuevo SHA y la referencia a actualizar. La primera línea indica también las capacidades disponibles en el cliente. A continuación, el cliente envía un archivo empaquetado con todos los objetos que faltan en el

servidor. Y, por ultimo, el servidor responde con un indicador de éxito (o fracaso) de la operación:

```
000Aunpack ok
```

### Descargando datos, (Downloading)

Cuando descargas datos, los procesos que se ven envueltos son **fetch-pack** (recuperar paquete) y **upload-pack** (enviar paquete). El cliente arranca un proceso **fetch-pack**, para conectar con un proceso **upload-pack** en el lado servidor y negociar con él los datos a transferir.

Hay varias maneras de iniciar un proceso **upload-pack** en el repositorio remoto. Se puede lanzar a través de SSH, de la misma forma que se arrancaba el proceso **receive-pack**. O se puede arrancar a través del demonio Git, que suele estar escuchando por el puerto 9418. Tras la conexión, el proceso **fetch-pack** envía datos de una forma parecida a esta:

```
003fgit-upload-pack schacon/simplegit-progit.git\0host=myserver.com\0
```

Como siempre, comienza con 4 bytes indicadores de cuantos datos siguen a continuación, siguiendo con el comando a lanzar, y terminando con un byte nulo, el nombre del servidor y otro byte nulo más. El demonio Git realizará las comprobaciones de si el comando se puede lanzar, si el repositorio existe y si tenemos permisos. Siendo todo correcto, el demonio lanzará el proceso **upload-pack** y procesará nuestra petición.

Si en lugar de utilizar el demonio Git, estás utilizando el protocolo SSH. **fetch-pack** lanzará algo como esto:

```
$ ssh -x git@github.com "git-upload-pack 'schacon/simplegit-progit.git'"
```

En cualquier caso, después de establecer conexión, **upload-pack** responderá:

```
0088ca82a6dff817ec66f44342007202690a93763949 HEAD\0multi_ack thin-pack \
  side-band side-band-64k ofs-delta shallow no-progress include-tag
003fca82a6dff817ec66f44342007202690a93763949 refs/heads/master
003e085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 refs/heads/topic
0000
```



La respuesta es muy similar a la dada por **receive-pack**, pero las capacidades que se indican son diferentes. Además, nos indica la referencia HEAD, para que el cliente pueda saber qué ha de activar (check out) en el caso de estar requiriendo un clon.

En este punto, el proceso **fetch-pack** revisa los objetos que tiene y responde indicando los objetos que necesita. Enviando "want" (quiero) y la clave SHA que necesita. Los objetos que ya tiene, los envía con "have" (tengo) y la correspondiente clave SHA. Llegando al final de la lista, escribe "done" (hecho). Para indicar al proceso **upload-pack** que ya puede comenzar a enviar el archivo empaquetado con los datos requeridos:

```
0054want ca82a6dff817ec66f44342007202690a93763949 ofs-delta
0032have 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
0000
0009done
```

Este es un caso muy sencillo para ilustrar los protocolos de transferencia. En casos más complejos, el cliente explota las capacidades de **multi\_ack** (múltiples confirmaciones) o **side-band** (banda lateral). Pero este ejemplo muestra los intercambios básicos empleados en los protocolos inteligentes.

## 9.7 Los entresijos internos de Git - Mantenimiento y recuperación de datos

### Mantenimiento y recuperación de datos

De vez en cuando, es posible que necesites hacer algo de limpieza, (compactar un repositorio, adecuar un repositorio importado, recuperar trabajo perdido,...). En ese apartado vamos a ver algunos de esos escenarios.

#### Mantenimiento

De cuando en cuando, Git lanza automáticamente un comando llamado "auto gc". La mayor parte de las veces, este comando no hace nada. Pero, cuando hay demasiados objetos sueltos, (objetos fuera de un archivo empaquetador), o demasiados archivos empaquetadores, Git lanza un comando **git gc** completo. **gc** corresponde a "recogida de basura" (garbage collect), y este

comando realiza toda una serie de acciones: recoge los objetos sueltos y los agrupa en archivos empaquetadores; consolida los archivos empaquetadores pequeños en un solo gran archivo empaquetador; retira los objetos antiguos no incorporados a ninguna confirmación de cambios.

También puedes lanzar "auto gc" manualmente:

```
$ git gc --auto
```

Y, habitualmente, no hará nada. Ya que es necesaria la presencia de unos 7.000 objetos sueltos o más de 50 archivos empaquetadores para que Git termine lanzando realmente un comando "gc". Estos límites pueden configurarse con las opciones de configuración `gc.auto` y `gc.autopacklimit`, respectivamente.

Otra tarea realizada por `gc` es el empaquetar referencias en un solo archivo. Por ejemplo, suponiendo que tienes las siguientes ramas y etiquetas en tu repositorio:

```
$ find .git/refs -type f
.git/refs/heads/experiment
.git/refs/heads/master
.git/refs/tags/v1.0
.git/refs/tags/v1.1
```

Lanzando el comando `git gc`, dejarás de tener esos archivos en la carpeta `refs`. En aras de la eficiencia, Git los moverá a un archivo denominado `.git/packed-refs`:

```
$ cat .git/packed-refs
# pack-refs with: peeled
cac0cab538b970a37ea1e769cbbde608743bc96d refs/heads/experiment
ab1afef80fac8e34258ff41fc1b867c702daa24b refs/heads/master
cac0cab538b970a37ea1e769cbbde608743bc96d refs/tags/v1.0
9585191f37f7b0fb9444f35a9bf50de191beadc2 refs/tags/v1.1
^1a410efbd13591db07496601ebc7a059dd55cfe9
```

Si actualizas alguna de las referencias, Git no modificará este archivo. Sino que, en cambio, escribirá uno nuevo en `refs/heads`. Para obtener la clave SHA correspondiente a una determinada referencia, Git comprobará primero en la carpeta `refs` y luego en el archivo `packed-refs`. Cualquier referencia que no puedas encontrar en la carpeta `refs`, es muy posible que la encuentres en el archivo `packed-refs`.

Merece destacar que la última línea de este archivo comenzaba con ^- Esto nos indica que la etiqueta inmediatamente anterior es una etiqueta anotada y que esa línea es la confirmación de cambios a la que apunta dicha etiqueta anotada.

## Recuperación de datos

En algún momento de tu trabajo con Git, perderás por error una confirmación de cambios. Normalmente, esto suele suceder porque has forzado el borrado de una rama con trabajos no confirmados en ella, y luego te has dado cuenta de que realmente necesitabas dicha rama; o porque has reculado (hard-reset) una rama, abandonando todas sus confirmaciones de cambio, y luego te has dado cuenta que necesitabas alguna de ellas. Asumiendo que estas cosas pasan, ¿cómo podrías recuperar tus confirmaciones de cambio perdidas?

Vamos a ver un ejemplo de un retroceso a una confirmación (commit) antigua en la rama principal de tu repositorio de pruebas, y cómo podríamos recuperar las confirmaciones perdidas en este caso. Lo primero es revisar el estado de tu repositorio en ese momento:

```
$ git log --pretty=oneline
ab1afef80fac8e34258ff41fc1b867c702daa24b modified repo a bit
484a59275031909e19aadb7c92262719cfcdf19a added repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Después, al mover la rama **master** de vuelta a la confirmación de cambios intermedia:

```
$ git reset --hard 1a410efbd13591db07496601ebc7a059dd55cfe9
HEAD is now at 1a410ef third commit
$ git log --pretty=oneline
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Vemos que se han perdido las dos últimas confirmaciones de cambios, --no tienes ninguna rama que te permita acceder a ellas--. Necesitas localizar el SHA de la última confirmación de cambios y luego añadir una rama que apunte hacia ella. El problema es cómo localizarla, --porque, ¿no te la sabrás de memoria, no?--.

El método más rápido para conseguirlo suele ser utilizar una herramienta denominada **git reflog**. Según trabajas, Git suele guardar un silencioso registro de donde está HEAD en cada

momento. Cada vez que confirmas cambios o cambias de rama, el registro (reflog) es actualizado. El registro reflog se actualiza incluso cuando utilizas el comando `git update-ref`. Siendo esta otra de las razones por las que es recomendable utilizar ese comando en lugar de escribir manualmente los valores SHA en los archivos de referencia, tal y como hemos visto anteriormente en la sección "Referencias Git". Con el comando `git reflog` puedes revisar donde has estado en cualquier momento pasado:

```
$ git reflog
1a410ef HEAD@{0}: 1a410efbd13591db07496601ebc7a059dd55cfe9: updating HEAD
ab1afef HEAD@{1}: ab1afef80fac8e34258ff41fc1b867c702daa24b: updating HEAD
```

Se pueden ver las dos confirmaciones de cambios que hemos activado, pero no hay mucha más información al respecto. Para ver la misma información de manera mucho más amigable, podemos utilizar el comando `git log -g`. Este nos muestra una salida normal de registro:

```
$ git log -g
commit 1a410efbd13591db07496601ebc7a059dd55cfe9
Reflog: HEAD@{0} (Scott Chacon <schacon@gmail.com>)
Reflog message: updating HEAD
Author: Scott Chacon <schacon@gmail.com>
Date: Fri May 22 18:22:37 2009 -0700
```

third commit

```
commit ab1afef80fac8e34258ff41fc1b867c702daa24b
Reflog: HEAD@{1} (Scott Chacon <schacon@gmail.com>)
Reflog message: updating HEAD
Author: Scott Chacon <schacon@gmail.com>
Date: Fri May 22 18:15:24 2009 -0700
```

modified repo a bit

Parece que la confirmación de cambios perdida es esta última. Puedes recuperarla creando una nueva rama apuntando a ella. Por ejemplo, puedes iniciar una rama llamada `recover-branch` con dicha confirmación de cambios (ab1afef):

```
$ git branch recover-branch ab1afef
$ git log --pretty=oneline recover-branch
ab1afef80fac8e34258ff41fc1b867c702daa24b modified repo a bit
```

```
484a59275031909e19aadb7c92262719cfcdf19a added repo.rb  
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit  
cac0cab538b970a37ea1e769cbbde608743bc96d second commit  
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

¡Bravo!, acabas de añadir una rama denominada **recover-branch** al punto donde estaba originalmente tu rama **master**; permitiendo así recuperar el acceso a las dos primeras confirmaciones de cambios.

A continuación, supongamos que la pérdida se ha producido por alguna razón que no se refleja en el registro (reflog). Puedes simularlo borrando la rama **recover-branch** y borrando asimismo el registro. Con eso, perdemos completamente el acceso a las dos primeras confirmaciones de cambio:

```
$ git branch -D recover-branch  
$ rm -Rf .git/logs/
```

La información de registro (reflog) se guarda en la carpeta **.git/logs/**; por lo que, borrarla, nos quedamos efectivamente sin registro. ¿Cómo podríamos ahora recuperar esas confirmaciones de cambio? Un camino es utilizando el comando de chequeo de integridad de la base de datos: **git fsck**. Si lo lanzas con la opción **--full**, te mostrará todos los objetos sin referencias a ningún otro objeto:

```
$ git fsck --full  
dangling blob d670460b4b4aece5915caf5c68d12f560a9fe3e4  
dangling commit ab1afef80fac8e34258ff41fc1b867c702daa24b  
dangling tree aea790b9a58f6cf6f2804eeac9f0abbe9631e4c9  
dangling blob 7108f7ecb345ee9d0084193f147cdad4d2998293
```

En este caso, puedes ver la confirmación de cambios perdida en la línea 'dangling commit.....'. Y la puedes recuperar del mismo modo, añadiendo una rama que apunte a esa clave SHA.

## **Borrando objetos**

Git tiene grandes cosas. Pero el hecho de que un **git clone** siempre descargue la historia completa del proyecto (incluyendo todas y cada una de las versiones de todos y cada uno de los archivos). Puede casar problemas. Todo suele ir bien si el contenido es únicamente código fuente. Ya que Git está tremendamente optimizado para comprimir eficientemente ese tipo de datos. Pero, si alguien, en cualquier momento de tu proyecto, ha añadido un solo archivo enorme. A partir de

ese momento, todos los clones, siempre, se verán obligados a copiar ese enorme archivo. Incluso si ya ha sido borrado del proyecto en la siguiente confirmación de cambios realizada inmediatamente tras la que lo añadió. Porque en algún momento formó parte del proyecto, siempre permanecerá ahí.

Esto suele dar bastantes problemas cuando estás convirtiendo repositorios de Subversion o de Perforce a Git. En esos sistemas, uno no se suele descargar la historia completa. Y, por tanto, los archivos enormes no tienen mayores consecuencias. Si, tras una importación de otro sistema, o por otras razones, descubres que tu repositorio es mucho mayor de lo que debería ser. Es momento de buscar y borrar objetos enormes en él.

Una advertencia importante: estas técnicas son destructivas y alteran el historia de confirmaciones de cambio. Se basan en reescribir todos los objetos confirmados aguas abajo desde el árbol más reciente modificado para borrar la referencia a un archivo enorme. No tendrás problemas si lo haces inmediatamente después de una importación; o justo antes de que alguien haya comenzado a trabajar con la confirmación de cambios modificada. Pero si no es el caso, tendrás que avisar a todas las personas que hayan contribuido. Porque se verán obligadas a reorganizar su trabajo en base a tus nuevas confirmaciones de cambio.

Para probarlo por tí mismo, puedes añadir un archivo enorme a tu repositorio de pruebas y retirarlo en la siguiente confirmación de cambios. Así podrás practicar la búsqueda y borrado permanente del repositorio. Para empezar, añade un objeto enorme a tu historial:

```
$ curl http://kernel.org/pub/software/scm/git/git-1.6.3.1.tar.bz2 > git.tbz2
$ git add git.tbz2
$ git commit -am 'added git tarball'
[master 6df7640] added git tarball
 1 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 git.tbz2
```

!Ouch!, --no querías añadir un archivo tan grande a tu proyecto--. Mejor si lo quitas:

```
$ git rm git.tbz2
rm 'git.tbz2'
$ git commit -m 'oops - removed large tarball'
[master da3f30d] oops - removed large tarball
 1 files changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 git.tbz2
```

Ahora, puedes limpiar **gc** tu base de datos y comprobar cuánto espacio estás ocupando:

```
$ git gc
Counting objects: 21, done.
Delta compression using 2 threads.
Compressing objects: 100% (16/16), done.
Writing objects: 100% (21/21), done.
Total 21 (delta 3), reused 15 (delta 1)
```

Puedes utilizar el comando **count-objects** para revisar rápidamente el espacio utilizado:

```
$ git count-objects -v
count: 4
size: 16
in-pack: 21
packs: 1
size-pack: 2016
prune-packable: 0
garbage: 0
```

El valor de **size-pack** nos da el tamaño de tus archivos empaquetadores, en kilobytes, y, por lo que se ve, estás usando 2 MB. Antes de la última confirmación de cambios, estabas usando algo así como 2 KB. Resulta claro que esa última confirmación de cambios no ha borrado el archivo enorme del historial. A partir de este momento, cada vez que alguien haga un clón de este repositorio, se verá obligado a copiar 2 MB para un proyecto tan simple. Y todo porque tu añadiste accidentalmente un archivo enorme en algún momento. Para arreglar la situación.

Lo primero es localizar el archivo enorme. En este caso, ya sabes de antemano cual es. Pero suponiendo que no lo supieras, ¿cómo podrías identificar el archivo o archivos que están ocupando tanto espacio?. Tras lanzar el comando **git gc**, todos los objetos estarán guardados en un archivo empaquetador. Puedes identificar los objetos enormes en su interior, utilizando otro comando de fontanería denominado **git verify-pack** y ordenando su salida por su tercera columna, la que nos informa de los tamaños de cada objeto. Puedes también redirigir su salida a través del comando **tail**. Porque realmente solo nos interesan las últimas líneas, las correspondientes a los archivos más grandes.

```
$ git verify-pack -v .git/objects/pack/pack-3f8c0...bb.idx | sort -k 3 -n | tail -3
e3f094f522629ae358806b17daf78246c27c007b blob 1486 734 4667
05408d195263d853f09dca71d55116663690c27c blob 12908 3478 1189
```

```
7a9eb2fba2b1811321254ac360970fc169ba2330 blob 2056716 2056872 5401
```

El archivo enorme es el último: 2 MB (2056716 Bytes para ser exactos). Para concretar cual es el archivo, puedes utilizar el comando **rev-list** que ya vimos brevemente en el capítulo 7. Con la opción **--objects**, obtendrás la lista de todas las SHA de todas las confirmaciones de cambio, junto a las SHA de los objetos binarios y las ubicaciones (paths) de cada uno de ellos. Puedes usar esta información para localizar el nombre del objeto binario:

```
$ git rev-list --objects --all | grep 7a9eb2fb
7a9eb2fba2b1811321254ac360970fc169ba2330 git.tbz2
```

Una vez tengas ese dato, lo puedes utilizar para borrar ese archivo en todos los árboles pasados. Es sencillo revisar cuales son las confirmaciones de cambios donde interviene ese archivo:

```
$ git log --pretty=oneline --branches -- git.tbz2
da3f30d019005479c99eb4c3406225613985a1db oops - removed large tarball
6df764092f3e7c8f5f94cbe08ee5cf42e92a0289 added git tarball
```

Para borrar realmente ese archivo de tu historial Git, has de reescribir todas las confirmaciones de cambio aguas abajo de **6df76**. Y, para ello, puedes emplear el comando **filter-branch** que se vió en el capítulo 6.

```
$ git filter-branch --index-filter \
    'git rm --cached --ignore-unmatch git.tbz2' -- 6df7640^..
Rewrite 6df764092f3e7c8f5f94cbe08ee5cf42e92a0289 (1/2)rm 'git.tbz2'
Rewrite da3f30d019005479c99eb4c3406225613985a1db (2/2)
Ref 'refs/heads/master' was rewritten
```

La opción **--index-filter** es similar a la **--tree-filter** vista en el capítulo 6. Se diferencia porque, en lugar de modificar archivos activados (checked out) en el disco, se modifica el área de preparación (staging area) o índice. En lugar de borrar un archivo concreto con una orden tal como **rm archivo**, has de borrarlo con **git rm --cached** (es decir, tienes que borrarlo del índice, en lugar de del disco). Con eso conseguimos aumentar la velocidad. El proceso es mucho más rápido, porque Git no ha de activar cada revisión a disco antes de procesar tu filtro. Aunque también puedes hacer lo mismo con la opción **--tree-filter**, si así lo prefieres. La opción **--ignore-unmatch** indica a **git rm** que evite lanzar errores en caso de no encontrar el patrón que le has ordenado buscar. Y por último, se indica a **filter-branch** que reescriba la historia a



partir de la confirmación de cambios `6df7640`, porque ya conocemos que es a partir de ahí donde comenzaba el problema. De otro modo, el comando comenzaría desde el principio, asumiendo un proceso innecesariamente más largo.

Tras esto, el historial ya no contiene ninguna referencia a ese archivo. Pero, sin embargo, quedan referencias en el registro (reflog) y en el nuevo conjunto de referencias en `.git/refs/original` que Git ha añadido al procesar `filter-branch`. Por lo que has de borrar también estás y reempaquetar la base de datos. Antes de reempaquetar, asegurate de acabar completamente con cualquier elemento que apunte a las viejas confirmaciones de cambios:

```
$ rm -Rf .git/refs/original
$ rm -Rf .git/logs/
$ git gc
Counting objects: 19, done.
Delta compression using 2 threads.
Compressing objects: 100% (14/14), done.
Writing objects: 100% (19/19), done.
Total 19 (delta 3), reused 16 (delta 1)
```

Y ahora, vamos a ver cuanto espacio hemos ahorrado.

```
$ git count-objects -v
count: 8
size: 2040
in-pack: 19
packs: 1
size-pack: 7
prune-packable: 0
garbage: 0
```

El tamaño del repositorio ahora es de 7 KB, mucho mejor que los 2 MB anteriores. Por el valor de "size", puedes ver que el objeto enorme sigue estando entre tus objetos sueltos; es decir, no hemos acabado completamente con él. Pero lo importante es que ya no será considerado al transferir o clonar el proyecto. Si realmente quieres acabar del todo, puedes lanzar la orden `git prune --expire` para retirar incluso ese archivo suelto.

BOOKS FOR PROFESSIONALS BY PROFESSIONALS®

## Pro Git

*Pro Git (Second Edition)* is your fully-updated guide to Git and its usage in the modern world. Git has come a long way since it was first developed by Linus Torvalds for Linux kernel development. It has taken the open source world by storm since its inception in 2005, and this book teaches you how to use it like a pro.

Effective and well-implemented version control is a necessity for successful web projects, whether large or small. With this book you'll learn how to master the world of distributed version workflow, use the distributed features of Git to the full, and extend Git to meet your every need.

Written by Git pros **Scott Chacon** and **Ben Straub**, *Pro Git (Second Edition)* builds on the hugely successful first edition, and is now fully updated for Git version 2.0, as well as including an indispensable chapter on GitHub. It's the best book for all your Git needs.

Shelve in  
Software Engineering/Software Development

User level:  
Beginning–Advanced

SOURCE CODE ONLINE



**Apress®**  
www.apress.com