Git

Fundamentos

Contenido

.1 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 familizarizado 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 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.

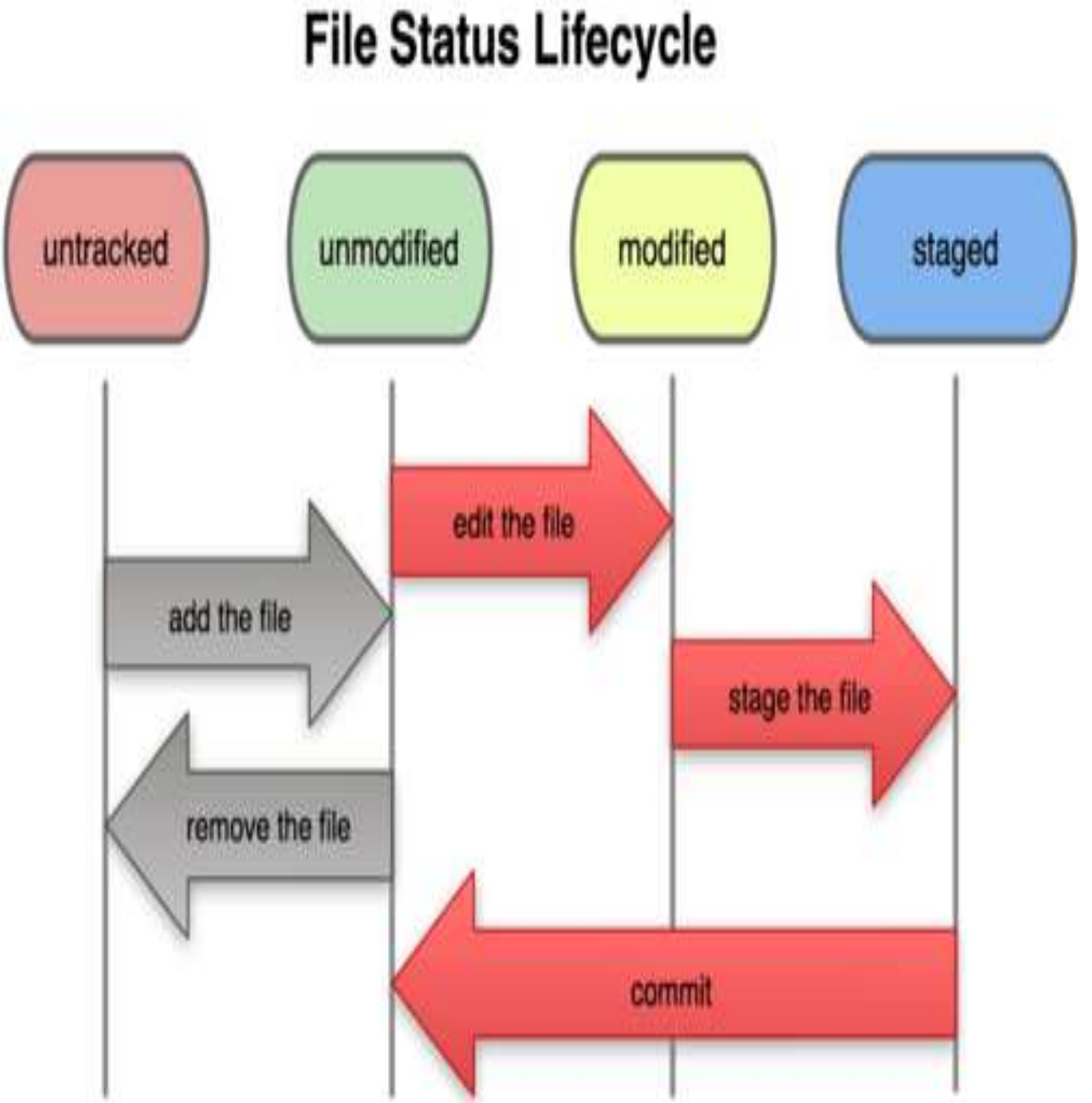

Figura 2-1. El ciclo de vida del estado de tus archivos.

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 0 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;

([0-9]) y caracteres entre corchetes separados por un guión 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 puede que quieras hacer es mantener el archivo en tu directorio de trabajo, pero eliminarlo de tu área de preparación. Dicho de otro modo, puede que quieras 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 explicito 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 renombras 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.

.3 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

Opción	Descripción de la salida
%h	Hash de la confirmación abreviado
%T	Hash del árbol
%t	Hash del árbol abreviado
%P	Hashes de las confirmaciones padre
%p	Hashes de las confirmaciones padre abreviados
%an	Nombre del autor
%ae	Dirección de correo del autor
%ad	Fecha de autoría (el formato respeta la opción --date)
%ar	Fecha de autoría, relativa

Opción	Descripción de la salida
%cn	Nombre del confirmador
%ce	Dirección de correo del confirmador
%cd	Fecha de confirmación
%cr	Fecha de confirmación, relativa
%s	Asunto

Puede que te estés preguntando la diferencia entre *autor* (*author*) y *confirmador* (*committer*). El autor es la persona que escribió originalmente el trabajo, mientras que el confirmador es quien lo aplicó. Por tanto, si mandas un parche a un proyecto, y uno de sus miembros lo aplica, ambos recibiréis

reconocimiento —tú como autor, y el miembro del proyecto como confirmador—. Veremos esta distinción en mayor profundidad en 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
-p	Muestra el parche introducido en cada confirmación.
--word-diff	Muestra el parche en formato de una palabra.
--stat	Muestra estadísticas sobre los archivos modificados en cada confirmación.
--shortstat	Muestra solamente la línea de resumen de la opción --stat.
--name-only	Muestra la lista de archivos afectados.
--name-status	Muestra la lista de archivos afectados, indicando además si

Opción	Descripción
	fueron añadidos, modificados o eliminados.
--abbrev-commit	Muestra solamente los primeros caracteres de la suma SHA-1, en vez de los 40 caracteres de que se compone.
--relative-date	Muestra la fecha en formato relativo (por ejemplo, “2 weeks ago” (“hace 2 semanas”)) en lugar del formato completo.
--graph	Muestra un gráfico ASCII con la historia de ramificaciones y uniones.

Opción	Descripción
--pretty	Muestra las confirmaciones usando un formato alternativo. Posibles opciones son oneline, short, full, fuller y format (mediante el cual puedes especificar tu propio formato).
--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
-(n)	Muestra solamente las últimas n confirmaciones
--since, --after	Muestra aquellas confirmaciones hechas después de la fecha especificada.
--until, --before	Muestra aquellas confirmaciones hechas antes de la fecha especificada.
--author	Muestra sólo aquellas confirmaciones cuyo autor coincide con la cadena especificada.

Opción	Descripción
--committer	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.

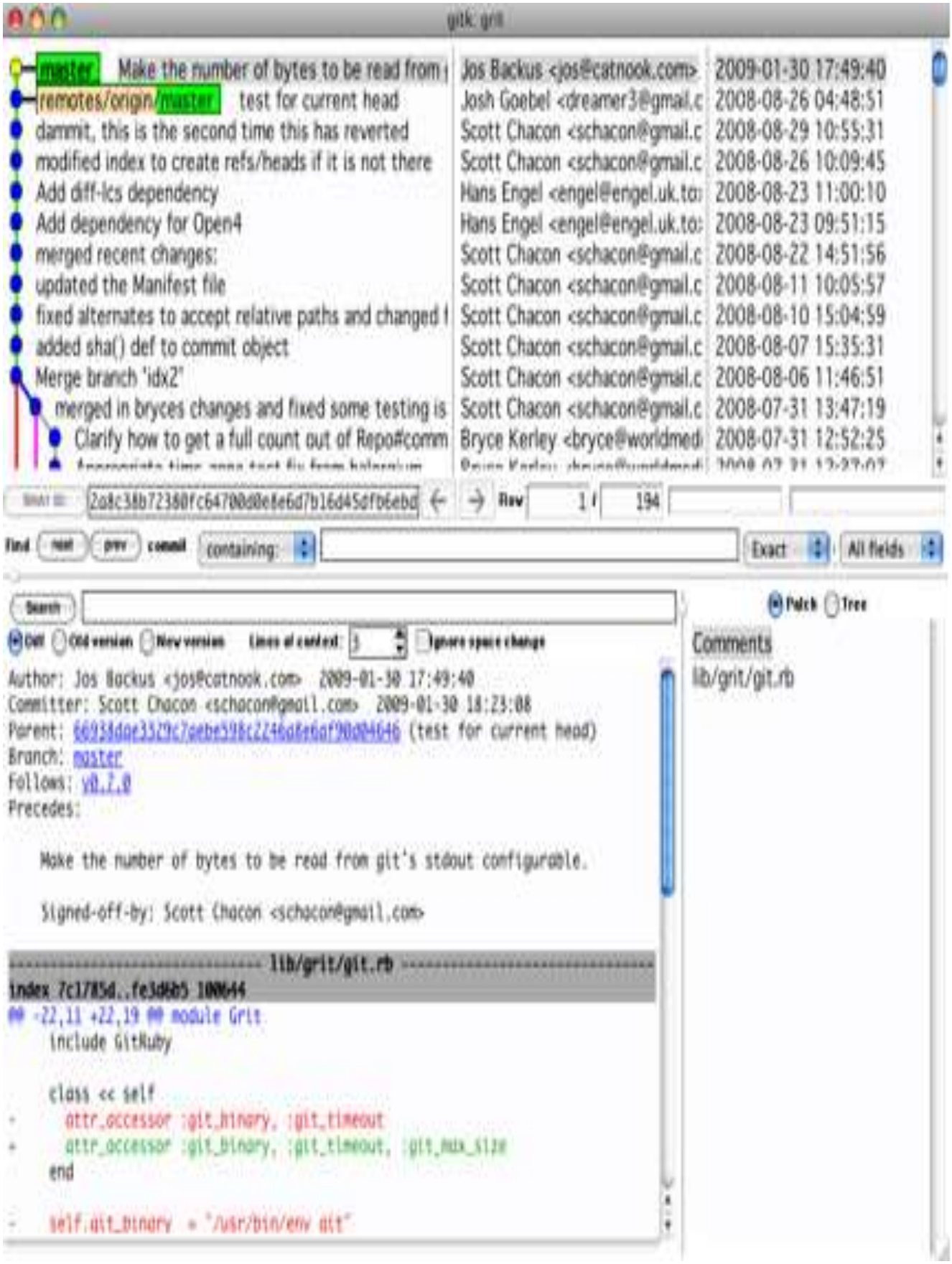

Figura 2-2. El visualizador de histórico gitk.

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.

.4 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 sobreescribirá 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 sobreescribirlo 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 sobreescritas 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.

5 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 clonais 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
```

.6 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 mas 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)

iEYEABECAAYFAkmQurIACgkQON3DxfchxFr5c

ACeIMN+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 explicitamente 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.

.7 Consejos y trucos

Antes de que terminemos este capitulo 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'

.8 Resumen

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