**Git**

* **Capitulo 1: Empezando**

**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.

# 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. En 2005, la relación entre la comunidad que desarrollaba el núcleo de Linux y la compañía que desarrollaba BitKeeper se vino abajo, Esto impulsó a la comunidad de desarrollo de Linux a desarrollar su propia herramienta. 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.

# 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. 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 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. 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, 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 se muestra.



Figura. 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. 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.

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.

### Los tres estados

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, añadiendolos 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).

* **Capitulo 2: Fundamentos de Git**

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. 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.

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'

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. Git recibe una copia de casi todos los datos que tiene el servidor. 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.

# 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.



Figura. 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

### 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.

# 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 |
| %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 |
| %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 fueron añadidos, modificados o eliminados. |
| --abbrev-commit | Muestra solamente los primeros caracteres de la suma SHA-1, en vez de los 40 caracteres de que se compone. |
| --relative-date | Muestra la fecha en formato relativo (por ejemplo, “2 weeks ago” (“hace 2 semanas”)) en lugar del formato completo. |
| --graph | Muestra un gráfico ASCII con la historia de ramificaciones y uniones. |
| --pretty | Muestra las confirmaciones usando un formato alternativo. Posibles opciones son oneline, short, full, fuller y format (mediante el cual puedes especificar tu propio formato). |
| --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. |
| --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.

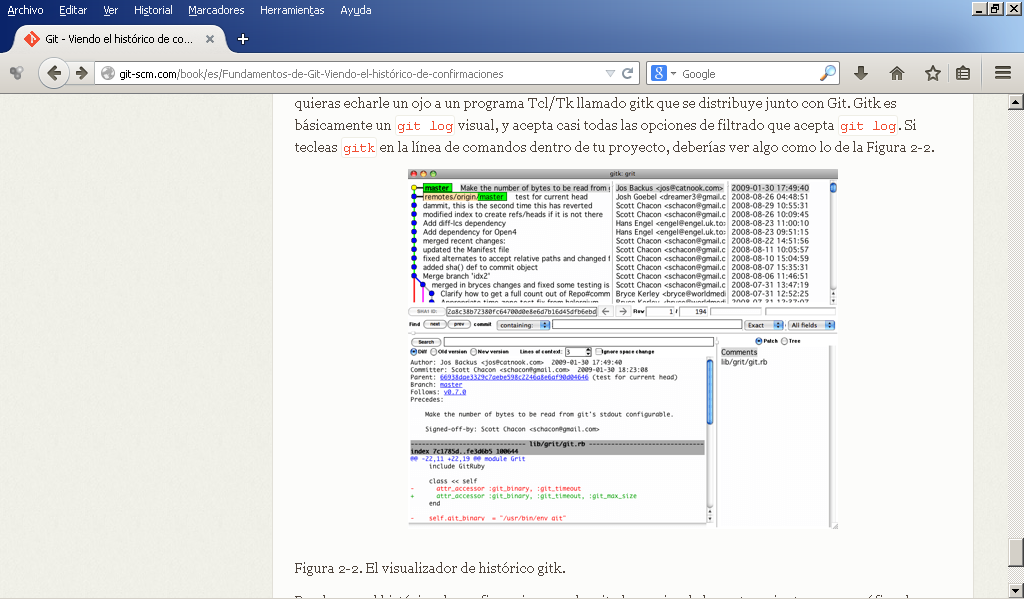


Figura 2-2. El visualizador de histórico gitk.

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.