Creando Repositorios GIT y GITHUB

Empezando - Una breve historia de Git

Una breve historia de Git

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

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

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

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

Empezando - Fundamentos de Git

Fundamentos de Git

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

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

Git tiene integridad

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

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

24b9da6552252987aa493b52f8696cd6d3b00373

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

Git generalmente sólo añade información

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

Esto hace que usar Git sea un placer, porque sabemos que podemos experimentar sin peligro de fastidiar gravemente las cosas. Para un análisis más exhaustivo de cómo almacena Git su información y cómo puedes recuperar datos aparentemente perdidos.

Los tres estados

Ahora presta atención. Esto es lo más importante a recordar acerca de Git si quieres que el resto de tu proceso de aprendizaje prosiga sin problemas. Git tiene tres estados principales en los que se pueden encontrar tus archivos: confirmado (committed), modificado (modified), y preparado (staged). Confirmado significa que los datos están almacenados de manera segura en tu base de datos local. Modificado significa que has modificado el archivo pero todavía no lo has confirmado a tu base de datos. Preparado significa que has marcado un archivo modificado en su versión actual para que vaya en tu próxima confirmación.

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

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.

Modificas una serie de archivos en tu directorio de trabajo.

Preparas los archivos, añadiéndolos a tu área de preparación.

Confirmas los cambios, lo que toma los archivos tal y como están en el área de preparación, y almacena esas instantáneas de manera permanente en tu directorio de Git.

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

Empezando - Instalando Git

Instalando Git

Vamos a empezar a usar un poco de Git. Lo primero es lo primero: tienes que instalarlo. Puedes obtenerlo de varias maneras; las dos principales son instalarlo desde código fuente, o instalar un paquete existente para tu plataforma.

Instalando desde código fuente

Si puedes, en general es útil instalar Git desde código fuente, porque obtendrás la versión más reciente. Cada versión de Git tiende a incluir útiles mejoras en la interfaz de usuario, por lo que utilizar la última versión es a menudo el camino más adecuado si te sientes cómodo compilando software desde código fuente. También ocurre que muchas distribuciones de Linux contienen paquetes muy antiguos; así que a menos que estés en una distribución muy actualizada o estés usando backports, instalar desde código fuente puede ser la mejor opción.

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

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

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

Cuando tengas todas las dependencias necesarias, puedes descargar la versión más reciente de Git desde su página web:

http://git-scm.com/download

Luego compila e instala:

```
$ tar -zxf git-1.6.0.5.tar.gz
$ cd git-1.6.0.5
$ make prefix=/usr/local all
$ sudo make prefix=/usr/local install
```

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

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

Instalando en Linux

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

```
$ yum install git-core
```

O si estás en una distribución basada en Debian como Ubuntu, prueba con apt-get:

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

Instalando en Mac

Hay tres maneras fáciles de instalar Git en un Mac. La más sencilla es usar el instalador gráfico de Git, que puedes descargar desde la página de SourceForge (véase Figura 1-7):

http://sourceforge.net/projects/git-osx-installer/

Una alternativa es instalar Git a través de MacPorts (http://www.macports.org). Si tienes MacPorts instalado, instala Git con:

```
$ sudo port install git-core +svn +doc +bash_completion +gitweb
```

No necesitas añadir todos los extras, pero probablemente quieras incluir +svn en caso de que alguna vez necesites usar Git con repositorios Subversión (véase el Capítulo 8).

La segunda alternativa es Homebrew (http://brew.sh/). Si ya tienes instalado Homebrew, instala Git con:

```
$ brew install git
```

Instalando en Windows

Instalar Git en Windows es muy fácil. El proyecto msysGit tiene uno de los procesos de instalación más sencillos. Simplemente descarga el archivo exe del instalador desde la página de GitHub, y ejecútalo:

http://msysgit.github.com/

Una vez instalado, tendrás tanto la versión de línea de comandos (incluido un cliente SSH que nos será útil más adelante) como la interfaz gráfica de usuario estándar.

Nota para el uso en Windows: Se debería usar Git con la shell provista por msysGit (estilo Unix), lo cual permite usar las complejas líneas de comandos de este libro. Si por cualquier razón se necesitara usar la shell nativa de Windows, la consola de línea de comandos, se han de usar las comillas dobles en vez de las simples (para parámetros que contengan espacios) y se deben entrecomillar los parámetros terminándolos con el acento circunflejo (^) si están al final de la línea, ya que en Windows es uno de los símbolos de continuación.

Empezando - Configurando Git por primera vez

Configurando Git por primera vez

Ahora que tienes Git en tu sistema, querrás hacer algunas cosas para personalizar tu entorno de Git. Sólo es necesario hacer estas cosas una vez; se mantendrán entre actualizaciones. También puedes cambiarlas en cualquier momento volviendo a ejecutar los comandos correspondientes.

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

Archivo /etc/gitconfig: Contiene valores para todos los usuarios del sistema y todos sus repositorios. Si pasas la opción --system a git config, lee y escribe específicamente en este archivo.

Archivo ~/.gitconfig file: Específico a tu usuario. Puedes hacer que Git lea y escriba específicamente en este archivo pasando la opción --global.

Archivo config en el directorio de Git (es decir, .git/config) del repositorio que estés utilizando actualmente: Específico a ese repositorio. Cada nivel sobrescribe los valores del nivel anterior, por lo que los valores de .git/config tienen preferencia sobre los de /etc/gitconfig.

En sistemas Windows, Git busca el archivo .gitconfig en el directorio $HOME (%USERPROFILE% in Windows' environment), que es C:\Documents and Settings\$USER para la mayoría de usuarios, dependiendo de la versión ($USER es %USERNAME% en el entorno Windows). También busca en el directorio /etc/gitconfig, aunque esta ruta es relativa a la raíz MSys, que es donde quiera que decidieses instalar Git en tu sistema Windows cuando ejecutaste el instalador.

Tu identidad

Lo primero que deberías hacer cuando instalas Git es establecer tu nombre de usuario y dirección de correo electrónico. Esto es importante porque las confirmaciones de cambios (commits) en Git usan esta información, y es introducida de manera inmutable en los commits que envías:

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

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

Tu editor

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

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

Tu herramienta de diferencias

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

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

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

Comprobando tu configuración

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

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

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

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

```
$ git config user.name
```

Empezando - Obteniendo ayuda

Obteniendo ayuda

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

```
$ git help <comando>
```

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

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

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

```
$ git help config
```

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

Fundamentos de Git - Obteniendo un repositorio Git

Obteniendo un repositorio Git

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

Inicializando un repositorio en un directorio existente

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

```
$ git init
```

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

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

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

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

Clonando un repositorio existente

Si deseas obtener una copia de un repositorio Git existente —por ejemplo, un proyecto en el que te gustaría contribuir— el comando que necesitas es git clone. Si estás 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.

1Git 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.

Fundamentos de Git - Guardando cambios en el repositorio

Guardando cambios en el repositorio

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

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

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

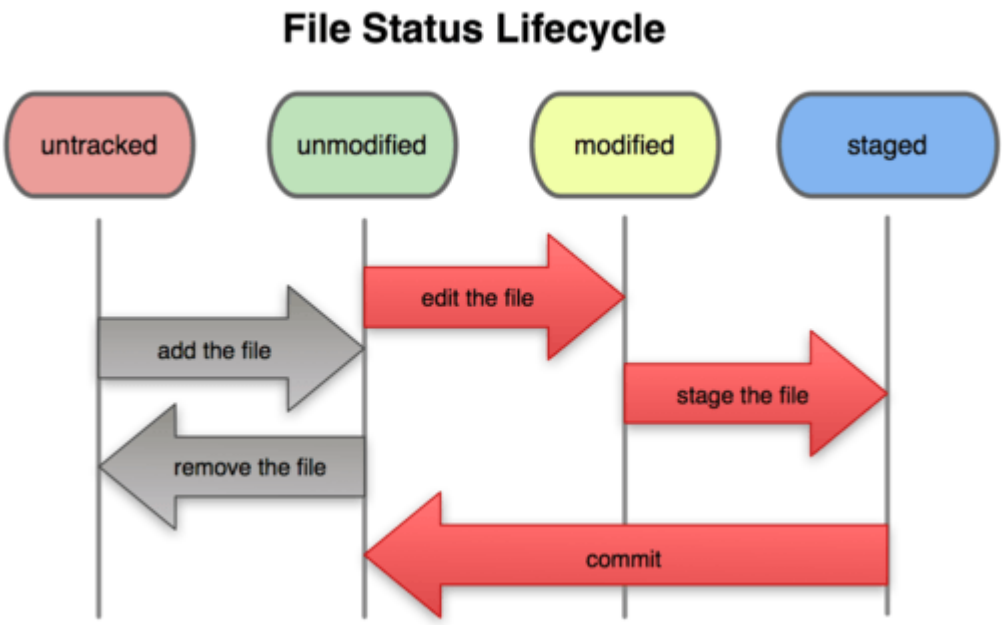

Figura 2-1. El ciclo de vida del estado de tus archivos.

Comprobando el estado de tus archivos

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

$ git status

On branch master

nothing to commit, working directory clean

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

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

```
$ vim README

$ git status

# On branch master

# Untracked files:

#   (use "git add <file>..." to include in what will be committed)

#

#   README

nothing added to commit but untracked files present (use "git add" to track)
```

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

Seguimiento de nuevos archivos

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

```
$ git add README
```

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

```
$ git status

# On branch master

# Changes to be committed:

#   (use "git reset HEAD <file>..." to unstage)

#
```

```
#   new file:   README
#
```

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

Preparando archivos modificados

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

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

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

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

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

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

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

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

Ignorando archivos

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

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

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

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

Las líneas en blanco, o que comienzan por #, son ignoradas.

Puedes usar patrones glob estándar.

Puedes indicar un directorio añadiendo una barra hacia delante (/) al final.

Puedes negar un patrón añadiendo una exclamación (!) al principio.

Los patrones glob son expresiones regulares simplificadas que pueden ser usadas por las shells. Un asterisco (*) reconoce cero o más caracteres; [abc] reconoce cualquier carácter de los especificados entre corchetes (en este caso, a, b o c); una interrogación (?) reconoce un único carácter; y caracteres entre corchetes separados por un guión ([0-9]) reconoce cualquier carácter entre ellos (en este caso, de 0 a 9).

He aquí otro ejemplo de archivo .gitignore:

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

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

Viendo tus cambios preparados y no preparados

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

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

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

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

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

```
      end

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

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

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

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

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

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

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

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

```
$ git diff
diff --git a/benchmarks.rb b/benchmarks.rb
index e445e28..86b2f7c 100644
--- a/benchmarks.rb
+++ b/benchmarks.rb
@@ -127,3 +127,4 @@ end
 main()
```

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

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

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

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

Confirmando tus cambios

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

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

```
$ git commit
```

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

El editor mostrará el siguiente texto (este ejemplo usa Vim):

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   README
#       modified:   benchmarks.rb
~
~
~
".git/COMMIT_EDITMSG" 10L, 283C
```

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

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

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

```
 create mode 100644 README
```

¡Acabas de crear tu primera confirmación! Puedes ver que el comando commit ha dado cierta información sobre la confirmación: a qué rama has confirmado (master), cuál es su suma de comprobación SHA-1 de la confirmación (463dc4f), cuántos archivos se modificaron, y estadísticas acerca de cuántas líneas se han añadido y cuántas se han eliminado.

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

Saltándote el área de preparación

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

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

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

Eliminando archivos

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

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

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

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

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

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

Otra cosa que 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.

Bibliografia

https://git-scm.com/book/es/v1/Fundamentos-de-Git-Obteniendo-un-repositorio-Git