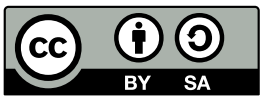
Apuntes de prácticas: Operaciones básicas con Git

Antonio García Domínguez
nyoescape arroba gmail punto com

11 de agosto de 2008



Índice

1. Configuración inicial

1. Primero tendremos que decirle a Git nuestro nombre y dirección de correo. Se utilizará para indicarnos como autor en cada una de nuestras revisiones y para firmar con GPG, entre otras cosas. Para ello, modificaremos la configuración global en `~/.gitignore` mediante estas órdenes:

```
git config --global user.name "Nombre Apellidos"
git config --global user.email micorreo@example.com
```

2. Ahora diremos a Git que nos coloree ciertas salidas de interés:

```
git config --global user.name
git config --global user.email
git config --global color.diff
```

3. Por último, indicaremos cuál es nuestro editor favorito (se usa Vim por defecto):

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

2. Inicialización de un repositorio

1. Para crear un repositorio vacío, en vez de clonar uno existente, lo que se hace es ejecutar `git init` sobre el directorio raíz con todos los ficheros que queramos controlar. Este repositorio no tendrá aún ninguna revisión enviada. Crearemos un directorio vacío en este caso y creamos el repositorio Git en su interior:

```
mkdir ejemplo-curso
git init
```

   Una nota: si el repositorio va a ser usado por varios usuarios, es recomendable utilizar la opción `--shared=group` para que todos los miembros del grupo al que pertenezcan los ficheros del repositorio puedan empujar a él.

2. Vemos que únicamente tenemos el directorio oculto `.git`:

```
ls -a
ls .git
```

   Este directorio contiene, entre otros, las entradas:

   config Fichero con la configuración local de Git para este repositorio.

description Descripción del repositorio para `gitweb`.

HEAD Referencia a la revisión actual con que trabajamos.

hooks Guiones Bash o programas para responder ante eventos.

info/exclude Fichero del estilo de `.gitignore` global para el repositorio y no controlado por Git. Lo veremos después.

objects Base de datos de objetos.

refs Referencias por nombre a las puntas de las distintas ramas locales y remotas y a otras revisiones etiquetadas por nombre.

3. Gestión básica de ficheros y revisiones

1. Ahora crearemos un fichero muy sencillo de dos líneas:

```
echo -e "A\nB" > f
```

2. Si miramos qué tal vamos con `git status`, obtendremos:

```
# On branch master
#
# Initial commit
#
# Untracked files:
# (use "git add <file>..." to include in what will be committed)
#
# f
nothing added to commit but untracked files present (use "git add" to
    track)
```

Con esto sabemos que:

- Estamos en la rama «master» de desarrollo. Esta es la rama principal estable que Git crea por defecto para todo repositorio. Veremos más tarde en profundidad el concepto y los usos de las ramas.
- La siguiente revisión que creemos será la primera revisión raíz del repositorio.
- Hay una serie de ficheros cuyos cambios no se están monitorizando: en este caso es el fichero `f` que antes creamos.

3. Para que Git monitorice `f`, lo añadimos a la zona intermedia de almacenamiento para preparar envíos (el *índice* o *caché*):

```
git add f
```

4. Si miramos de nuevo la salida de `git status`, veremos cómo ha cambiado su estado, indicando que el nuevo fichero `f` pasará a formar parte de la siguiente revisión:

```
# Changes to be committed:
# (use "git rm --cached <file>..." to unstage)
#
# new file: f
#
```

5. Ahora lo enviamos y creamos la primera revisión. Al ejecutar la siguiente orden, se abrirá nuestro editor y podremos introducir un mensaje para describir qué introduce la nueva revisión. Se recomienda que la primera línea sea un resumen corto de 50 o menos caracteres.

```
git commit
```

6. Ya se ha creado la primera revisión:

```
Created initial commit 2600adf: Primera revisión.
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 f
```

7. Ahora haremos unas cuantas revisiones más. La opción `-m` de `git commit` nos ahorra tener que darle un mensaje a cada nueva revisión:

```
cp f g
git add g
git commit -m "Copiado f a g"
echo B >> f
```

8. Si hicieramos `git status` ahora, veríamos:

```
# Changed but not updated:
# (use "git add <file>..." to update what will be committed)
#
# modified: f
```

   Tenemos una serie de cambios en el directorio de trabajo sobre `f`, pero no los hemos enviado al índice aún, así que todavía no son parte de la siguiente revisión. Para confirmar estos cambios para la próxima revisión, usaremos:

```
git add f
```

9. Seguimos con otras órdenes. La opción `-v` nos dará un informe de los cambios exactos que introduciremos en esta nueva revisión, que no formarán parte de su descripción.

```
cp f h
git add h
git commit -v
```

10. Es el momento de consultar nuestros logs, y comparar las salidas de:

```
git log --stat
git log -C --stat
git log -C --find-copies-harder --stat
```

    La primera salida no informa de ninguna copia: aparentemente, los ficheros `f`, `g` y `h` no guardan ninguna relación. Esto se debe a que Git únicamente sigue su contenido, y no guarda historiales de cada fichero, al estilo de Subversion, por ejemplo. Recordemos que Git es conocido como un «stupid content tracker», o monitor estúpido de contenido.

    Sin embargo, la segunda operación sí detecta la copia de `f` a `h`, pero no la de `f` a `g`. Hemos activado la detección heurística de copias, pero por defecto sólo se comparan aquellos ficheros que hayan sido cambiados en esa revisión, por motivos de rendimiento. Retirando esa limitación como hacemos en la tercera operación conseguimos que se detecte la copia.

11. Vamos ahora a practicar el borrado y renombrado de ficheros. De nuevo, a diferencia de Subversion, no hay nada especial que hacer. Primero añadiremos algunos ficheros en una nueva revisión y luego los moveremos:

```
echo X > mueveme
echo Y > borrame
git add mueveme borrame
git commit -m "Añadido ficheros para mover y borrar"
git rm borrame
mv mueveme movido
git add movido
git commit -a -m "Borrado un fichero y movido otro"
```

    Las tres últimas órdenes son más interesantes: primero, `git rm` retira un fichero de la copia de trabajo y del índice, marcando su borrado para la próxima revisión. Sin embargo, no hemos usado la orden `git mv` existente para mover `mueveme`, y en su lugar hemos pasado la opción `-a` a `git commit`.

    Esta opción añade todos los cambios pendientes sobre los ficheros que se hallen bajo control de versiones: con él, nos podemos ahorrar tener que manualmente enviar el `git rm mueveme` pertinente. Realmente, `git mv a b` es lo mismo que:

```
cp a b
git rm a
git add b
```

12. Ahora probaremos con:

```
git log −1 −−stat
git log −1 −M −−stat
```

La segunda orden activa la detección heurística de renombrados: se basa en que en una misma revisión se borre un fichero y aparezca otro con la misma huella SHA-1.

4. Uso avanzado del índice

4.1. Enviando cambios parciales

1. El uso de un índice como zona intermedia de preparación nos permite hacer muchas cosas que de otra forma serían muy complicadas de hacer. Una cosa que podemos hacer es enviar en una revisión sólo una parte de los cambios que hemos introducido en las líneas de un fichero. Probaremos a hacer lo siguiente:

```
echo A >> f
echo D | cat − f > fnuevo
mv fnuevo f
git add −p f
```

2. Se nos mostrará algo así:

```
diff −−git a/f b/f
index b1e6722..4e43a4f 100644
−−− a/f
+++ b/f
@@ −1,3 +1,5 @@
+D
 A
 B
 C
+A
Stage this hunk [y/n/a/d/s/?]?
```

3. Pediremos con «s» que nos divida este bloque o *hunk* en otros más pequeños, ya que queremos enviar sólo el cambio representado por la primera línea.

4. Ahora se nos mostrará el bloque con la primera línea, y diremos que sí lo prepare para su envío con «y».

5. Por último, respondiendo «n» al segundo bloque evitaremos que lo prepare para envío.

6. Ahora podríamos enviarlo, pero no lo haremos aún: nos servirá para la siguiente parte.

Una nota: los bloques sólo pueden dividirse de esta forma hasta el punto en que sean todos líneas contiguas. Para poder ir a un nivel más fino aún, tendremos que esperar a Git 1.6.0 y la nueva orden «e» para editar el bloque a mano. También se ha añadido la posibilidad de preparar una única línea para envío.

4.2. Comparando cambios realizados

1. Recordemos que en Git trabajamos con tres estructuras de datos: el repositorio (el directorio `.git`), el índice o caché (`.git/index`) y el directorio de trabajo (todo lo que está fuera de `.git`). Git nos permite hacer comparaciones entre los tres.

   Primero, una curiosidad: si ejecutamos `git status`, veremos algo así:

   ```
   # Changes to be committed:
   # (use "git reset HEAD <file>..." to unstage)
   #
   # modified: f
   #
   # Changed but not updated:
   # (use "git add <file>..." to update what will be committed)
   #
   # modified: f
   #
   ```

   La razón por la que aparece tanto para enviar como no actualizado es que sólo hemos preparado para envío algunos de los cambios introducidos. La primera parte viene de que la versión en el índice difiere de la de la última revisión, y la segunda de que la versión del índice también difiere de la del directorio de trabajo.

2. Primero, probaremos `git diff`, que obtiene las diferencias entre el directorio de trabajo y la versión preparada para envío en el índice. Esta orden destacará aquella línea que no preparamos para envío.

3. `git diff --cached` compara la versión del índice con la versión de la revisión actual (`HEAD`), y así nos señalará la línea que tenemos preparada para enviar.

4. `git diff HEAD` compara el directorio de trabajo con la revisión actual, y destacará por lo tanto las dos líneas que hemos introducido.

5. Ya podemos crear la siguiente revisión:

```
git commit −m "Añadido una línea al principio de f"
```

4.3. Deshaciendo cambios en el directorio de trabajo

1. Git nos permite deshacer los cambios que tengamos hechos sobre el índice o el directorio de trabajo de múltiples maneras. En primer lugar, podemos restaurar un fichero del directorio de trabajo a la versión que se halle en el índice. Tras el último `git commit` que hicimos, podemos descartar esa línea de `f` que en su momento no enviamos con:

```
git checkout f
```

   Si hiciéramos `git status` ahora, veríamos que nos dice que el directorio de trabajo se halla limpio (sin cambios pendientes). Veremos cómo hacerlo posteriormente.

2. Otra posibilidad es restaurar un fichero al estado que tenía en alguna otra revisión, que identificaremos con un *commitish*, con la misma sintaxis de los *treeish* pero resultando en una revisión en vez de un árbol. En particular, `HEAD^` apunta a la revisión anterior a la actual. Usaremos una variante de la orden anterior:

```
git checkout HEAD^ f
```

   Dado que tanto el índice como el directorio de trabajo son modificados, `git status` reflejará ahora que está listo para enviar en la siguiente revisión una copia de `f` con los contenidos que tenía hace 2 revisiones.

3. Supongamos que queremos cancelar este cambio que acabamos de hacer. Para retirar la versión modificada del índice y situar la existente en HEAD, usaremos:

```
git reset −− f
```

   Esta orden no revierte la copia del directorio de trabajo. Tendremos que hacer tal y como se vio en el primer paso:

```
git checkout f
```

4. Podemos usar `git reset` también para cancelar un borrado:

```
git rm h
git reset −− h
git checkout h
```

Acción a deshacer	Afecta a	Orden
Añadido Modificación Borrado	índice	`git reset -- f`
Añadido	directorio de trabajo	`rm f`
Modificación Borrado		`git checkout f`
Añadido	ambos de una vez	`git rm -f f`
Modificación Borrado		`git checkout HEAD f`

Cuadro 1: Formas de deshacer cambios sobre el fichero `f`

Si queremos reunir las dos últimas órdenes en una sola, podemos hacer lo siguiente:

```
git checkout HEAD h
```

Esto recupera de una sola vez tanto la entrada del índice (segunda orden) como la copia del directorio de trabajo (tercera orden).

5. `git reset` también sirve para cancelar el añadido de un nuevo fichero:

```
touch j
git add j
git reset −− j
rm j
```

Al igual que antes, al hacer `git reset` no se toca el nuevo `j` de la copia de trabajo, así que lo tendremos que borrar a mano. Se pueden hacer los dos últimos pasos de una sola orden con `git rm -f j`.

Todas las posibilidades se hallan reunidas en la tabla 1.

5. Corrigiendo errores en revisiones anteriores

Si no hemos enviado aún nuestros cambios a ningún otro repositorio ni nadie los ha tomado de nuestro repositorio, podemos corregir errores fácilmente en la última revisión que hayamos hecho. Simplemente haremos como si preparáramos una nueva revisión y al enviarla utilizar la opción `--amend`:

```
git commit −−amend
```

Con esto, la anterior revisión se ignorará y será efectivamente sustituida por otra que incorporará sus cambios y los que hayamos hecho ahora.

Si ya hemos enviado a alguien la revisión con las equivocaciones, no podemos hacer esto. En su lugar, tenemos `git revert <commitish>` que genera una nueva revisión que invierte los cambios de la revisión señalada por el *commitish* proporcionado. Esto también es útil para revisiones anteriores a la actual: por ejemplo, podríamos deshacer los cambios hechos en `HEAD~2` con `git revert HEAD^`.

Puede que queramos arreglar una o varias revisiones sin dejar rastro: por ejemplo, para después enviar una serie de revisiones a una lista de correo y no molestarles con nuestras equivocaciones. Una posible solución de bajo nivel (realmente recomendamos usar `git rebase --interactive`, descrito en un documento posterior) es emplear `git reset` para mover el puntero `HEAD` a una revisión anterior. Para movernos al primer padre del primer padre de `HEAD` y hacer corresponder el índice, el índice y el directorio de trabajo o ninguno de los dos, usaríamos `git reset HEAD~2`, `git reset --hard HEAD~2` o `git reset --soft HEAD~2`, respectivamente, y seguiríamos trabajando desde allí.

El `HEAD` antiguo y su padre son efectivamente olvidados, y serán borrados automáticamente al ejecutar `git gc` tras un período de gracia (por defecto de 2 semanas) o inmediatamente al ejecutar `git prune`.

La única forma de acceder a esas revisiones olvidadas es mediante la sintaxis `HEAD@{N}`, donde N es un entero, que señala al N-ésimo valor anterior que tomó `HEAD`. Esto no salvará a esas revisiones de ser «limpiadas»: para retirarlas de todo peligro, hay que asegurarse de que sean accesibles desde una etiqueta o la última revisión o *punta* («tip» en el original inglés) de alguna rama. Veremos más tarde en este documento cómo etiquetar cualquier revisión, y se describirá en un documento posterior el proceso de creación de ramas.

Repetimos igualmente que *no* debemos modificar revisiones que hayan sido enviadas a algún otro repositorio si queremos poder colaborar entre ambos.

6. Etiquetando revisiones

Cada revisión se puede identificar por un *commitish* de manera inequívoca. Hay varias opciones, mejor descritas bajo la página *man* de `git-rev-parse`, como usar un prefijo único de su huella SHA-1 o alguna ruta relativa a otro *commitish*, usando ciertas relaciones de ascendencia y demás.

Muchas veces, sin embargo, nos interesará darle un nombre más descriptivo, como «v1.0», por ejemplo. Estas referencias por nombres arbitrarios se conocen como *etiquetas*, y se pueden crear de una forma muy sencilla.

1. Podemos etiquetar la revisión actual con:

```
git tag v2.0
```

Como no hemos pasado otro argumento que el nombre de la etiqueta, se etiquetará la revisión actual con una etiqueta «ligera», que sólo incluye el SHA-1 de la revisión a la que referencia.

2. También podemos etiquetar cualquier otra revisión dando su *commitish*, y crear objetos reales de tipo etiqueta con comentarios (`-a`) y firmadas con nuestra clave pública GnuPG (`-s`) que tenga el mismo correo y nombre que le dimos a Git:

```
git tag −a −s v1.0 HEAD~2
```

3. Para listarlas todas, podemos usar simplemente:

```
git tag
```

4. Para verificarlas, usaríamos, por ejemplo:

```
git tag −v v1.0
```

Deberíamos de ver algo así al final:

```
gpg: Firmado el mié 06 ago 2008 18:40:26 CEST usando clave DSA ID 73
     D6A764
gpg: Firma correcta de "Antonio García <nyoescape@gmail.com>"
gpg: alias "Antonio Garcia <nyoescape@gmail.com>"
```

5. Para mover una etiqueta, usamos la opción `-f` para forzar su sobreescritura. Obviamente, esto sólo lo podemos hacer si aún nadie tiene una copia de esta etiqueta:

```
git tag −f v2.0 HEAD~3
```

6. Para retirar una etiqueta, podemos usar la opción `-d`:

```
git tag −d v1.0
```

7. Un detalle adicional: si quisiéramos saber en qué versión está basada la revisión anterior, por ejemplo, podríamos usar esta orden:

```
git describe HEAD^
```

Por defecto sólo tiene en cuenta los objetos etiqueta y no las etiquetas ligeras. Podemos corregir esto con la opción `--all`.

7. Distribuyendo revisiones

Podemos redistribuir los contenidos de cualquier revisión mediante la orden `git archive`. Si, por ejemplo, quisiéramos distribuir la versión 1.0 en formato `tar.gz`, usaríamos:

```
git archive v1.0 | gzip > ../miproyecto−1.0.tar.gz
```

Puede que nos interese añadir un cierto prefijo a la ruta de todos los ficheros (por ejemplo, para que se descomprima dentro de un subdirectorio). Para ello tenemos la opción `--prefix`. Es muy importante que no se nos olvide la barra final:

```
git archive v1.0 −−prefix=miproyecto−1.0/ | gzip > ../miproyecto−1.0.tar.gz
```

Por otro lado, podríamos distribuir la revisión anterior a la actual en formato `zip` (se usa `tar` por defecto) con:

```
git archive −−format=zip HEADˆ > ../miproyecto−1.0.zip
```

8. Mantenimiento de un repositorio Git

A diferencia de otros sistemas como Subversion o Mercurial, un repositorio Git tiene la pega de requerir cierto mantenimiento periódico para ahorrar espacio y tiempo de ejecución. Sin embargo, no se trata de un problema grave: sólo hemos de ejecutar `git gc` de vez en cuando para eliminar objetos inútiles y aprovechar mejor el espacio, entre otras tareas de mantenimiento. De todas formas, podemos hacerlo todo también de forma manual.

Para comprobar la consistencia de nuestro repositorio y ver si hay algún objeto que deje de ser alcanzable por alguna de las ramas, etiquetas o el registro de valores de las referencias o *reflog*, usaremos `git fsck`. Con la opción `--no-reflogs` podemos no tener en cuenta el *reflog* y así obtener resultados más precisos de lo que realmente está en nuestro repositorio.

Una de las cosas que `git gc` hace es retirar los objetos a los que no se puede llegar desde ninguna referencia con nombre que no sea `HEAD`, como una etiqueta o la punta de alguna rama de desarrollo. Esto lo hace tras un período de gracia configurable de 30 días por defecto, pero se puede ejecutar inmediatamente para todos los objetos sueltos (como fichero individuales en `.git/objects`) del repositorio mediante `git prune`. Los objetos empaquetados no son retirados con `git prune`: en su lugar, tendremos que ejecutar `git repack -a -d`, que reunirá todos los objetos y *packs* alcanzables en un solo nuevo *pack* y borrará todos los que queden inalcanzables.