GIT – Guía rápida

Notas:

- *Cambios* locales se refiere a los ficheros del directorio de trabajo (*working dir*) que hayan sido modificados desde el último commit.
- *COMMIT* es un indicador cualquiera en el repositorio: SHA1 de un commit, tag, HEAD (último commit de la rama actual), HEAD~1 (antecesor de HEAD)... La mayoría de las veces si no se indica se asume HEAD (último commit de la rama actual)

Enlaces:

- http://git-scm.org
- http://www-cs-students.stanford.edu/~blynn/gitmagic/intl/es/ch02.html

Inicializar repositorio

```
$ git init
$ git add .                         # Marca todos los ficheros para el próximo commit.
$ git commit -m "Estado inicial"    # Crea un nuevo commit con los ficheros marcados.
```

Crear ramas

Una rama es un puntero a un commit específico en el repositorio.

```
$ git branch <nombre>               # Crear la rama en el punto actual. Es necesario hacer checkout a la misma.
$ git branch <nombre> <COMMIT>      # Crea la rama a partir del commit dado. Es necesario hacer checkout.
$ git checkout -b <nombre>          # Crear rama en el punto actual y hacerle checkout.
$ git checkout -b <nombre> <COMMIT> # Crear la rama a partir del commit dado y hacerle checkout.
$ git branch -m <actual> <nuevo>    # Renombrar la rama
$ git branch -d <nombre>            # Borrar la rama
```

Al crear una rama nueva y hacerle checkout los cambios locales se trasladan a esa rama, con lo que el siguiente commit será sobre la rama nueva.

Listar ramas

```
$ git branch                        # Listar todas las ramas
$ git branch -v                     # Mostrar último commit en cada rama y su situación respecto a su rama remota (si hay)
$ git branch --merged               # Mostrar ramas que se han fusionado con la actual, y por tanto pueden borrarse
$ git branch --no-merged            # Mostrar ramas con trabajos sin fusionar. Intentar borrarlas dará un error.
```

Moverse a una rama o a un commit específico

```
$ git checkout <COMMIT>             # No toca los cambios locales
$ git checkout -f <COMMIT>          # Sobreescribe los cambios locales
```

Se puede hacer una rama desde el commit actual para continuar el desarrollo (git branch). Si se hacen cambios en un commit intermedio (no es un *HEAD*) y se commitean sin hacer una rama, se crea un commit separado que sale del actual (rama sin nombre).

Fusionar ramas (merge)

```
$ git merge <nombre>                # Fusiona la rama indicada en la rama actual
```

Las diferencias se resuelven automáticamente si es posible. En caso de conflictos (código o ficheros binarios modificados en ambas ramas) el proceso se detiene (*merging*) a la espera de una resolución manual.

Resolver conflictos de fusionado:

Dentro de cada fichero en conflicto se añaden marcas alrededor del código conflictivo, mostrando al mismo tiempo la versión de una y otra rama (excepto en ficheros binarios)

```
$ git status                        # Muestra la situación actual del merge (Unmerged paths)
$ git diff                          # Muestra los ficheros conflictivos y las diferencias
$ git add <file>                    # Marca el fichero como corregido una vez resuelto el conflicto
$ git rm <file>                     # Marca el fichero como eliminado en la revisión resuelta
```

La sección "Unmerged paths" de *git status* muestra los ficheros que reclaman atención. Debe resolverse cada conflicto manualmente dentro del fichero (eliminando las marcas agregadas por git) y marcarlo como resuelto con *git add*.

En vez de editar los ficheros es posible escoger una de las dos versiones disponibles (rama actual o rama que se está fusionando):

```
$ git checkout --ours -- <file>     # Obtener la versión del fichero en la rama actual
$ git checkout --theirs  <file>     # Obtener la versión del fichero en la rama que se está fusionando con la actual
```

Para abortar la acción o deshacerla una vez realizada:

```
$ git reset --hard HEAD             # Abortar el proceso y volver a la situación anterior al intento de merge
$ git reset --hard ORIG_HEAD        # Deshacer si ya se había confirmado con git commit
```

Una vez resueltos todos los conflictos se confirma el proceso:

```
$ git commit                        # Confirmar la fusión (merge) una vez resueltos todos los conflictos
```

Resolución gráfica de conflictos:

```
$ git mergetool                     # Inicia la herramienta gráfica de resolución de conflictos
```

La herramienta crea ficheros adicionales por cada fichero en conflicto (backup, base, local, remote) para que la herramienta de resolución pueda mostrarlos al usuario al mismo tiempo y éste establecer la versión final. Estos ficheros deberían borrarse automáticamente tras la edición (en caso de que persistan es necesario borrarlos manualmente).

La resolución básica sólo sirve para ficheros de texto. En ficheros binarios usar *git checkout –ours* o *git checkout –theirs* para escoger una de las dos versiones disponibles.

Configurar una herramienta gráfica para resolver conflictos:
http://www.davesquared.net/2010/03/easier-way-to-set-up-diff-and-merge.html (Windows)
http://gitguru.com/2009/02/22/integrating-git-with-a-visual-merge-tool (Mac)

Deshacer cambios

```
$ git revert <COMMIT>             # Deshacer de forma segura los cambios introducidos por un commit cualquiera
$ git reset --hard                # Deshace los cambios locales
$ git reset --hard HEAD~1         # Elimina el último commit
```

Recuperar una versión determinada de un fichero o path:

```
$ git reset <COMMIT> -- <path>    # git reset NO sobreescribe cambios locales
$ git reset -p <COMMIT> -- <path> # Seleccionar interactivamente las partes a restaurar
$ git checkout <COMMIT> -- <path> # Sobreescribe cambios locales sin preguntar
```

En Windows se puede abrir git-bash directamente en cualquier subcarpeta carpeta del proyecto (boton derecho – git bash here). Entonces para recuperar un fichero o path local

```
$ git checkout <COMMIT> -- ./<path>
```

Conocer el historial de un fichero

```
$ git log <path>                  # Mostrar todos los commits para un fichero especifico con info detallada
$ git log -n 2    <path>          # Mostrar sólo los dos últimos commits para ese fichero
$ git log -oneline -- <path>      # Formato abreviado con id de commit y comentario
```

```
$ git log <SINCE>..<UNTIL> -- <path>  # Mostrar los commits para ese fichero entre dos commits indicados
```

Abrir GTK mostrando gráficamente el historial para un fichero o ruta dado:

```
$ gitk <path>
```

Localizar y restaurar ficheros borrados

```
$ git log --diff-filter=D --summary     # Mostrar los ficheros borrados en los últimos commits
$ git checkout <COMMIT>^ -- <path>      # Restaurar un fichero borrado en un commit dado
```

El ^ al final del commit es para restaurar el fichero desde el commit anterior al que fue borrado. Equivale a <commit>~1.

Guardar cambios actuales para recuperarlos después

Guarda los cambios desde el último commit. Al recuperarlos, si hay colisiones se hace un *merge*.

```
$ git stash               # Guarda cambios hechos desde el ultimo commit
$ git stash pop           # Recupera los cambios guardados
$ git stash list          # Lista los estados guardados
$ git stash apply         # Aplica cambios guardados sin borrarlos de la lista
```

Marcar el commit actual (Tag)

```
$ git tag -s <nombre> -m <mensaje>
```

El tag queda firmado usando la firma GPG asociada al autor (ver Creating SSH keys).

El nombre identifica al tag y se usa en los demás comandos (ej. git checkout). Por ejemplo: v2.32.45r1

```
$ git tag                         # Mostrar lista de tags
$ git tag -n                      # Mostrar lista y descripción
```

```
$ git tag -d <nombre>             # Eliminar Tag
$ git tag -a <nombre>             # Crear Tag no firmado
$ git push --tags                 # Subir Tags al repositorio remoto
$ git push origin :refs/tags/<nombre>  # Eliminar Tag borrado localmente
```

Localizar ficheros con una cadena de texto

```
$ git grep <texto>                # Mira en todos los ficheros del repositorio
$ git grep <texto> -- <ruta>      # Mira sólo en la ruta o rutas especificadas
                                  # Admite patrones (ej. *.cpp)
```

Trabajo con repositorios remotos

Obtener el repositorio desde otra localización (*fork*):

```
$ git clone <ruta al repositorio>     # Clonar y hacer checkout del HEAD de la rama actual
$ git clone -n <ruta al repositorio   # Clonar pero no hacer checkout
```

No hay que hacer *git init* ni crear directorio (se crea automáticamente a partir de la carpeta actual). La ruta puede ser una carpeta local, carpeta en red, URL, o cualquier otra referencia a un repositorio remoto. Si es privado será necesario tener la clave SSH configurada adecuadamente (ejemplo).

Recibir los cambios desde el repositorio original:

```
$ git pull
```

Es equivalente a

```
$ git fetch                       # Trae los cambios
$ git merge origin                # Fusionarlos con la versión actual
```

Subir cambios al repositorio:

```
$ git push origin <branch>        # Subir sólo la rama indicada
$ git push --all                  # Subir y actualizar todas las referencias remotas
```

Gestionar Tags en el repositorio:

```
$ git push --tags                  # Subir Tags (no suben de otra forma)
$ git push origin :refs/tags/<nombre>  # Eliminar Tag borrado localmente
```

Borrar una rama remota:

```
$ git push origin :<branch>
$ git push origin --delete <branch>   # GIT versión 1.7.0+
```

Listar los repositorios remotos y sus URLs:

```
$ git remote -v show
```

Cambiar la URL de un repositorio remoto (cambia ambos fetch y push)

```
$ git remote set-url origin <nueva URL>
```

Revertir un commit en local y también en el repositorio remoto:

(lo lógico es que no hubiera sido subido)

```
$ git reset --hard HEAD~1
$ git push origin +master:master
```

"The +master:master thing is necessary to tell git that you really do want to rewind the history here, (it's definitely not part of the normal flow)."

Tutorial

Push and delete branches

Tareas especiales

Marcar para commit sólo determinadas partes de un fichero

```
$ git add --patch          # Preguntar individualmente por cada cambio en todos los ficheros modificados
$ git add --patch <file>   # Preguntar individualmente por cada cambio en un fichero determinado
```

Se muestra cada cambio en el fichero individualmente y se pregunta qué hacer con él. Respuestas inmediatas:

```
y     aceptar este cambio para que entre en el próximo commit.
n     no marcar este cambio. No entrará en el próximo commit.
q     no marcar este cambio y salir. Se mantendrán los que ya se hayan marcado pero no los restantes.
a     marcar este cambio y todos los demás de este fichero.
d     no marcar este cambio ni ninguno de los restantes de este fichero.
```

Otras respuestas disponibles permiten saltar entre los cambios (g j k), reducirlos a partes más pecueñas (s), o editarlos manualmente (e).

Añadir un nuevo fichero o patrón a .gitignore

Añadirlo a *.gitignore* sigue controlando aquellos ficheros que ya están en el repositorio. Y

```
git rm <file>
```

eliminarlo el fichero del directorio de trabajo.

Para dejar de controlar el fichero o patrón manteniendo las copias actuales añadirlo a .gitignore y entonces:

```
$ git rm --cached <file>
$ git rm --cached -r <pattern>       # Eliminar ocurrencias en todo el arbol
```

Borrar el fichero completamente del repositorio implica reescribir toda la historia. Nada recomendable

Localizar el cambio que originó un problema

```
$ git bisect start
$ git bisect bad                 # La versión actual va mal
$ git bisect good v2.6.13-rc2    # esta versión es buena
```

Información

Manual para git bisect

Trabajar con submódulos (*submodules*)

Los submódulos son carpetas dentro de un repositorio cuyo contenido es a su vez un repositorio de GIT autogestionado

Añadir un submódulo a un repositorio

```
$ git submodule add <ruta al repositorio> <carpeta>       # Añade el repositorio dado como submódulo
                                                          # en la carpeta indicada del repositorio actual.
```

El submódulo se inicializa en la carpeta indicada y se le hace checkout a la rama Master

Los cambios locales en el repositorio anfitrión se limitan al fichero *.gitmodules* y a la nueva carpeta, pero no al contenido (*untracked content*). Desde el punto de vista del repositorio anfitrión un submódulo sólo consta de una definición en el fichero .gitmodules y de una carpeta. Esta carpeta es especial y hace referencia a un *commit* determinado en el origen del submódulo.

La carpeta con el submódulo es un repositorio GIT independiente. Se pueden usar comandos GIT desde esa carpeta, o *git gui* / *gitk* en ella. Los cambios afectarán sólo al contenido de la misma. Es posible crear y usar ramas, sincronizar con el origen, etc.

Notas:

- El repositorio a añadir como submódulo debe tener contenido (no sirve un repositorio recién creado, sin commits).
- Si agregar el submódulo produce alguno de estos fallos:

  fatal: Not a git repository: <carpeta>/../.git/<carpeta>

  The following path is ignored by one of your .gitignore files: <carpeta>

La solución es: (fuente)

```
$ rm -fr <carpeta> .git/modules/<carpeta>
```

Actualizar los submódulos

El repositorio anfitrión gestiona **la referencia** a un *commit* en el origen del submódulo, pero no gestiona el contenido de ese commit. Esta referencia se puede mover entre ramas, clonar, sincronizar con el origen, etc. Esto significa, por ejemplo, que al clonar un repositorio los submódulos estarán vacíos. El valor de la referencia sólo cambia (aparece en *git status*) cuando el submódulo se hace apuntar a un *commit* diferente en el origen. Por ejemplo, como resultado de commits locales en el submódulo, cambios de ramas, o sincronización con el origen.

Cuando la referencia cambia por operaciones en el repositorio anfitrión (ej. cambios de rama o *checkout* a revisiones anteriores), es necesario recrear el contenido apropiado dentro del submódulo para que coincida con el *commit* al que apuntaba en la revisión actual del anfitrión.

```
$ git submodule update --init              # Clona los submódulos que falten por clonar y hace checkout del commit referenciado.
                                           # --init asegura que los submódulos estén inicializados (ej. tras un clone).
$ git submodule update --init --recursive  # Si los submódulos contienen otros submódulos, actualizarlos a su vez.
```

El contenido de los submódulos se actualiza y gestiona con las operaciones GIT normales dentro del submódulo.

Mover un submódulo a otra carpeta del repositorio

Asegurarse que el repositorio y los submódulos están actualizados. Entonces seguir las instrucciones (fuente).

Ejemplo: mover un submódulo desde "Datos" a "var/Datos".

1. Editar .gitmodules cambiando name y path a "var/Datos"
2. Mover la carpeta GIT del submódulo desde ".git/modules/Datos" a ".git/modules/var/Datos"
3. Mover la carpeta del submódulo desde "Datos" a "var/Datos"
4. Editar ".git/modules/var/Datos/config" y corregir la línea
   ```
   [core] worktree
   ```
   . En este ejemplo cambia de
   ```
   ../../../Datos
   ```
   a
   ```
   ../../../../var/Datos
   ```
5. Editar "ext/Datos/.git" y corregir
   ```
   gitdir
   ```
   Aquí cambia de
   ```
   ../.git/modules/Datos
   ```
   a
   ```
   ../../git/modules/var/Datos
   ```
6. `git add .gitmodules`
7. `git rm --cached Datos`
8. `git submodule add -f <url> var/Datos`

En git status aparece

```
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   modified:   .gitmodules
#   renamed:    Datos -> var/Datos
```

En el último paso podría aparecer alguno de los fallos descritos al final del apartado *Añadir un submódulo a un repositorio*. En ese caso aplicar la misma resolución y repetir el paso 8.

Hacer un submódulo de una carpeta existente en un repositorio

El nuevo repositorio con el submódulo conservará el historial de los cambios en los ficheros de la carpeta (Fuente 1, Fuente 2).

1. Usar *git subtree* para crear una nueva rama que sólo contendrá a esa carpeta

```
$ git subtree split -P <carpeta> -b <nueva-rama>
```

2. Borrar la carpeta en la rama actual del repositorio

```
$ git rm -rf <carpeta>
```

3. Recrear la carpeta e inicializar un nuevo repositorio GIT en ella. El repositorio remoto para el submódulo debe estar ya creado

```
$ mkdir <carpeta>
$ pushd <carpeta>
$ git init
$ git remote add origin <ruta-al-repositorio-submodulo>
```

4. Traer (*pull*) al nuevo repositorio el contenido de <nueva-rama> en el repositorio anfitrión, y subir al repositorio remoto.

```
$ git pull ../ <nueva-rama>              # usar ../../ ó ../../../ etc. según la profundidad de <carpeta>
$ git push origin -u master
```

../ es la carpeta raíz del repositorio anfitrión. Si <carpeta> tiene varios niveles de profundidad será necesario encadenar ../../ ó ../../../ etc hasta llegar a la raíz del anfitrión

5. Añadir <carpeta> como submódulo al repositorio anfitrión

```
$ popd
$ git submodule add <ruta-al-repositorio-submodulo> <carpeta>
```

Hacer *commit* de los cambios para completar el proceso.

Eliminar un submódulo de un repositorio

Debe hacerse manualmente, no hay un comando para ello:

1. Eliminar la entrada del submódulo en el fichero *.gitmodules*. Si sólo hay un submódulo se puede borrar el fichero entero. Ejemplo:

```
[submodule "Assets/Common"]
path = Assets/Common
url = ssh://git@gitserver.com/user/assets-common.git
```

2. Eliminar la entrada del submódulo en el fichero .git/config. Ejemplo:

```
[submodule "Assets/Common"]
url = ssh://git@gitserver.com/user/assets-common.git
```

3. Borrar del índice la carpeta del submódulo

```
$ git rm --cached <carpeta>
```

4. Borrar las carpetas del submódulo en en el repositorio anfitrión

```
$ rm -fr <carpeta> .git/modules/<carpeta>
```

Hecho. El submódulo queda completamente eliminado del repositorio.

8 Comments

Fernando
Posted November 16, 2013 at 12:20 AM

Buena guía. Felicidades!!

leonardo
Posted December 8, 2013 at 6:15 AM

amigo, sabes como revertir un pull?

Edy
Posted September 12, 2014 at 7:32 PM

Leonardo, debería bastar con establecer el HEAD de la rama o ramas al commit que tenían antes del pull.

RJzue
Posted November 7, 2014 at 4:33 PM

Excelente! Gracias!

Luis
Posted November 26, 2015 at 4:48 PM

Muy buena Guía, Muchas gracias

Trackbacks / Pings

1. cosas buenas de git | gatotecherope
2. cosas buenas de git | Gatadas
3. Tutorial de git | tutorialeslibelula

Leave a Reply

Your email address will not be published. Required fields are marked *

Name *

Email *

Website

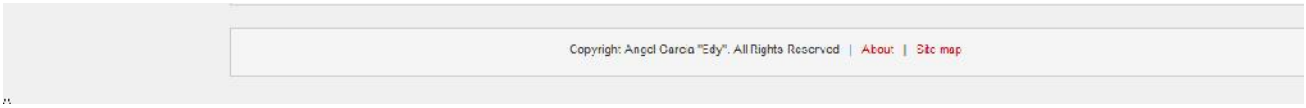
Confirm that you are not a bot - select a man with raised hand:

Comment

Post Comment

☐ Notify me of follow-up comments by email.

☐ Notify me of new posts by email.


Copyrigh: Angel Garcia "Edy". All Rights Reserved | Abou: | Site map