
This book is available in English.

Full translation available in

azərbaycan dili,
български език,
Deutsch,
Español,
Français,
Ελληνικά,
日本語,
한국어,
Nederlands,
Русский,
Slovenščina,
Tagalog,
Українська
简体中文,

Partial translations available in

Čeština,
Македонски,
Polski,
Српски,
Ўзбекча,
繁體中文,

Translations started for

Беларуская,
فارسی,

Indonesian,
Italiano,
Bahasa Melayu,
Português (Brasil),
Português (Portugal),
Svenska,
Türkçe.

The source of this book is hosted on GitHub.
Patches, suggestions and comments are welcome.

Chapters ▾

2nd Edition

2.2 Fundamentos de Git - Guardando cambios en el Repositorio

Guardando cambios en el Repositorio

Ya tienes un repositorio Git y un *checkout* o copia de trabajo de los archivos de dicho proyecto. El siguiente paso es realizar algunos cambios y confirmar instantáneas de esos cambios en el repositorio cada vez que el proyecto alcance un estado que quieras conservar.

Recuerda que cada archivo de tu repositorio puede tener dos estados: rastreados y sin rastrear. Los archivos rastreados (*tracked files* en inglés) son todos aquellos archivos que estaban en la última instantánea del proyecto; pueden ser archivos sin modificar, modificados o preparados. Los archivos sin rastrear son todos los demás - cualquier otro archivo en tu directorio de trabajo que no estaba en tu última instantánea y que no está en el área de preparación (*staging area*). Cuando clonas por primera vez un repositorio, todos tus archivos estarán rastreados y sin modificar pues acabas de sacarlos y aun no han sido editados.

Mientras editas archivos, Git los ve como modificados, pues han sido cambiados desde su último *commit*. Luego preparas estos archivos modificados y finalmente confirmas todos los cambios preparados, y repites el ciclo.

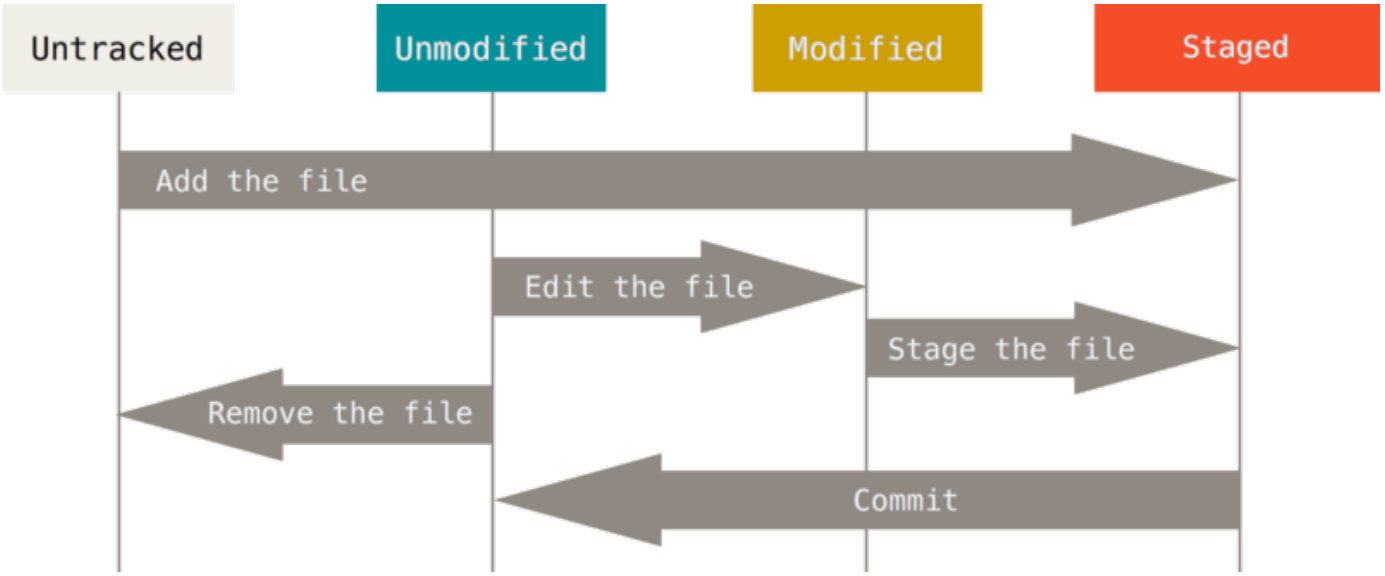

Figura 8. El ciclo de vida del estado de tus archivos.

Revisando el Estado de tus Archivos

La herramienta principal para determinar qué archivos están en qué estado es el comando `git status`. Si ejecutas este comando inmediatamente después de clonar un repositorio, deberías ver algo como esto:

```
$ git status
On branch master
nothing to commit, working directory clean
```

Esto significa que tienes un directorio de trabajo limpio - en otras palabras, que no hay archivos rastreados y modificados. Además, Git no encuentra archivos sin rastrear, de lo contrario aparecerían listados aquí. Finalmente, el comando te indica en cuál rama estás y te informa que no ha variado con respecto a la misma rama en el servidor. Por ahora, la rama siempre será “master”, que es la rama por defecto; no le prestaremos atención de momento. [ch03-git-branching] tratará en detalle las ramas y las referencias.

Supongamos que añades un nuevo archivo a tu proyecto, un simple README. Si el archivo no existía antes y ejecutas `git status`, verás el archivo sin rastrear de la siguiente manera:

```
$ echo 'My Project' > 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 el archivo README está sin rastrear porque aparece debajo del encabezado “Untracked files” (“Archivos no rastreados” en inglés) en la salida. Sin rastrear significa que Git ve archivos que no tenías en el *commit* anterior. Git no los incluirá en tu próximo *commit* a menos que se lo indiques explícitamente. Se comporta así para evitar incluir accidentalmente archivos binarios o cualquier otro archivo que no quieras incluir. Como tú sí quieres incluir README, debes comenzar a rastrearlo.

Rastrear Archivos Nuevos

Para comenzar a rastrear un archivo debes usar el comando `git add`. Para comenzar a rastrear el archivo README, puedes ejecutar lo siguiente:

```
$ git add README
```

Ahora si vuelves a ver el estado del proyecto, verás que el archivo README está siendo rastreado y está preparado para ser confirmado:

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
```

Puedes ver que está siendo rastreado porque aparece luego del encabezado “Cambios a ser confirmados” (“Changes to be committed” en inglés). Si confirmas en este punto, se guardará en el historial la versión del archivo correspondiente al instante en que ejecutaste `git add`. Anteriormente cuando ejecutaste `git init`, ejecutaste luego `git add (files)` - lo cual inició el rastreo de archivos en tu directorio. El comando `git add` puede recibir tanto una ruta de archivo como de un directorio; si es de un directorio, el comando añade recursivamente los archivos que están dentro de él.

Preparar Archivos Modificados

Vamos a cambiar un archivo que esté rastreado. Si cambias el archivo rastreado llamado “CONTRIBUTING.md” y luego ejecutas el comando `git status`, verás algo parecido a esto:

```
$ 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)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

El archivo “CONTRIBUTING.md” aparece en una sección llamada “Changes not staged for commit” (“Cambios no preparado para confirmar” en inglés) - lo que significa que existe un archivo rastreado que ha sido modificado en el directorio de trabajo pero que aún no está preparado. Para prepararlo, ejecutas el comando `git add`. `git add` es un comando que cumple varios propósitos - lo usas para empezar a rastrear archivos nuevos, preparar archivos, y hacer otras cosas como marcar archivos en conflicto por combinación como resueltos. Es más útil que lo veas como un comando para “añadir este contenido a la próxima confirmación” más que para “añadir este archivo al proyecto”. Ejecutemos `git add` para preparar el archivo “CONTRIBUTING.md” y luego ejecutemos `git status`:

```
$ git add CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
    modified:   CONTRIBUTING.md
```

Ambos archivos están preparados y formarán parte de tu próxima confirmación. En este momento, supongamos que recuerdas que debes hacer un pequeño cambio en `CONTRIBUTING.md` antes de confirmarlo. Abres de nuevo el archivo, lo cambias y ahora estás listos para confirmar. Sin embargo, ejecutemos `git status` una vez más:

```
$ vim CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
    modified:   CONTRIBUTING.md

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

¡¿Pero qué…?! Ahora `CONTRIBUTING.md` aparece como preparado y como no preparado. ¿Cómo es posible? Resulta que Git prepara un archivo de acuerdo al estado que tenía cuando ejecutas el comando `git add`. Si confirmas ahora, se confirmará la versión de `CONTRIBUTING.md` que tenías la última vez que ejecutaste `git add` y no la versión que ves ahora en tu directorio de trabajo al ejecutar `git status`. Si modificas un archivo luego de ejecutar `git add`, deberás ejecutar `git add` de nuevo para preparar la última versión del archivo:

```
$ git add CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
    modified:   CONTRIBUTING.md
```

Estado Abreviado

Si bien es cierto que la salida de `git status` es bastante explícita, también es verdad que es muy extensa. Git ofrece una opción para obtener un estado abreviado, de manera que puedas ver tus cambios de una forma más compacta. Si ejecutas `git status -s` o `git status --short`, obtendrás una salida mucho más simplificada.

```
$ git status -s
 M README
MM Rakefile
A  lib/git.rb
M  lib/simplegit.rb
?? LICENSE.txt
```

Los archivos nuevos que no están rastreados tienen un `??` a su lado, los archivos que están preparados tienen una `A` y los modificados una `M`. El estado aparece en dos columnas - la columna de la izquierda indica el estado preparado y la columna de la derecha indica el estado sin preparar. Por ejemplo, en esa salida, el archivo `README` está modificado en el directorio de trabajo pero no está preparado, mientras que `lib/simplegit.rb` está modificado y preparado. El archivo `Rakefile` fue modificado, preparado y modificado otra vez por lo que existen cambios preparados y sin preparar.

Ignorar Archivos

A veces, tendrás algún tipo de archivo que no quieres que Git añada automáticamente o más aun, que ni siquiera quieras que aparezca como no rastreado. Este suele ser el caso de archivos generados automáticamente como trazas o archivos creados por tu sistema de compilación. En estos casos, puedes crear un archivo llamado `.gitignore` que liste patrones a considerar. Este es un ejemplo de un archivo `.gitignore`:

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

La primera línea le indica a Git que ignore cualquier archivo que termine en “.o” o “.a” - archivos de objeto o librerías que pueden ser producto de compilar tu código. La segunda línea le indica a Git que ignore todos los archivos que terminen con una tilde (~), la cual es usada por varios editores de texto como Emacs para marcar archivos temporales. También puedes incluir cosas como trazas, temporales, o pid directamente; documentación generada automáticamente; etc. Crear un archivo `.gitignore` antes de comenzar a trabajar es generalmente una buena idea, pues así evitas confirmar accidentalmente archivos que en realidad no quieres incluir en tu repositorio Git.

Las reglas sobre los patrones que puedes incluir en el archivo `.gitignore` son las siguientes:

- Ignorar las líneas en blanco y aquellas que comiencen con `#`.
- Emplear patrones glob estándar que se aplicarán recursivamente a todo el directorio del repositorio local.
- Los patrones pueden comenzar en barra (`/`) para evitar recursividad.

- Los patrones pueden terminar en barra (`/`) para especificar un directorio.
- Los patrones pueden negarse si se añade al principio el signo de exclamación (`!`).

Los patrones glob son una especie de expresión regular simplificada usada por los terminales. Un asterisco (`*`) corresponde a cero o más caracteres; `[abc]` corresponde a cualquier caracter dentro de los corchetes (en este caso a, b o c); el signo de interrogación (`?`) corresponde a un caracter cualquiera; y los corchetes sobre caracteres separados por un guión (`[0-9]`) corresponde a cualquier caracter entre ellos (en este caso del 0 al 9). También puedes usar dos asteriscos para indicar directorios anidados; `a/**/z` coincide con `a/z`, `a/b/z`, `a/b/c/z`, etc.

Aquí puedes ver otro ejemplo de un archivo `.gitignore`:

```
# ignora los archivos terminados en .a
*.a

# pero no lib.a, aun cuando había ignorado los archivos terminados en .a en la línea anterior
!lib.a

# ignora unicamente el archivo TODO de la raiz, no subdir/TODO
/TODO

# ignora todos los archivos del directorio build/
build/

# ignora doc/notes.txt, pero no este: doc/server/arch.txt
doc/*.txt

# ignora todos los archivos .txt del directorio doc/
doc/**/*.txt
```

Sugerencia: GitHub mantiene una extensa lista de archivos `.gitignore` adecuados a docenas de proyectos y lenguajes en https://github.com/github/gitignore, en caso de que quieras tener un punto de partida para tu proyecto.

Ver los Cambios Preparados y No Preparados

Si el comando `git status` es muy impreciso para ti - quieres ver exactamente que ha cambiado, no solo cuáles archivos lo han hecho - puedes usar el comando `git diff`. Hablaremos sobre `git diff` más adelante, pero lo usarás probablemente para responder estas dos preguntas: ¿Qué has cambiado pero aun no has preparado? y ¿Qué has preparado y está listo para confirmar? A pesar de que `git status` responde a estas preguntas de forma muy general listando el nombre de los archivos, `git diff` te muestra las líneas exactas que fueron añadidas y eliminadas, es decir, el parche.

Supongamos que editas y preparas el archivo `README` de nuevo y luego editas `CONTRIBUTING.md` pero no lo preparas. Si ejecutas el comando `git status`, verás algo como esto:

```
$ 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)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

Para ver qué has cambiado pero aun no has preparado, escribe `git diff` sin más parámetros:

```
$ git diff
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 8ebb991..643e24f 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -65,7 +65,8 @@ branch directly, things can get messy.
 Please include a nice description of your changes when you submit your PR;
```

```
 if we have to read the whole diff to figure out why you're contributing
 in the first place, you're less likely to get feedback and have your change
-merged in.
+merged in. Also, split your changes into comprehensive chunks if you patch is
+longer than a dozen lines.

 If you are starting to work on a particular area, feel free to submit a PR
 that highlights your work in progress (and note in the PR title that it's
```

Este comando compara lo que tienes en tu directorio de trabajo con lo que está en el área de preparación. El resultado te indica los cambios que has hecho pero que aun no has preparado.

Si quieres ver lo que has preparado y será incluido en la próxima confirmación, puedes usar `git diff --staged`. Este comando compara tus cambios preparados con la última instantánea confirmada.

```
$ git diff --staged
diff --git a/README b/README
new file mode 100644
index 0000000..03902a1
--- /dev/null
+++ b/README
@@ -0,0 +1 @@
+My Project
```

Es importante resaltar que al llamar a `git diff` sin parámetros no verás los cambios desde tu última confirmación - solo verás los cambios que aun no están preparados. Esto puede ser confuso porque si preparas todos tus cambios, `git diff` no te devolverá ninguna salida.

Pasemos a otro ejemplo, si preparas el archivo `CONTRIBUTING.md` y luego lo editas, puedes usar `git diff` para ver los cambios en el archivo que ya están preparados y los cambios que no lo están. Si nuestro ambiente es como este:

```
$ git add CONTRIBUTING.md
$ echo 'test line' >> CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   CONTRIBUTING.md

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

Puedes usar `git diff` para ver qué está sin preparar

```
$ git diff
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 643e24f..87f08c8 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -119,3 +119,4 @@ at the
 ## Starter Projects

 See our [projects list](https://github.com/libgit2/libgit2/blob/development/PROJECTS.md).
+# test line
```

y `git diff --cached` para ver que has preparado hasta ahora (--staged y --cached son sinónimos):

```
$ git diff --cached
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 8ebb991..643e24f 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -65,7 +65,8 @@ branch directly, things can get messy.
 Please include a nice description of your changes when you submit your PR;
 if we have to read the whole diff to figure out why you're contributing
 in the first place, you're less likely to get feedback and have your change
-merged in.
+merged in. Also, split your changes into comprehensive chunks if you patch is
+longer than a dozen lines.
```

```
If you are starting to work on a particular area, feel free to submit a PR
that highlights your work in progress (and note in the PR title that it's
```

Git Diff como Herramienta Externa

Nota

A lo largo del libro, continuaremos usando el comando `git diff` de distintas maneras. Existe otra forma de ver estas diferencias si prefieres utilizar una interfaz gráfica u otro programa externo. Si ejecutas `git difftool` en vez de `git diff`, podrás ver los cambios con programas de este tipo como Araxis, emerge, vimdiff y más. Ejecuta `git difftool --tool-help` para ver qué tienes disponible en tu sistema.

Confirmar tus Cambios

Ahora que tu área de preparación está como quieres, puedes confirmar tus cambios. Recuerda que cualquier cosa que no esté preparada - cualquier archivo que hayas creado o modificado y que no hayas agregado con `git add` desde su edición - no será confirmado. Se mantendrán como archivos modificados en tu disco. En este caso, digamos que la última vez que ejecutaste `git status` verificaste que todo estaba preparado y que estás listo para confirmar tus cambios. La forma más sencilla de confirmar es escribiendo `git commit`:

```
$ git commit
```

Al hacerlo, arrancará el editor de tu preferencia. (El editor se establece a través de la variable de ambiente `$EDITOR` de tu terminal - usualmente es vim o emacs, aunque puedes configurarlo con el editor que quieras usando el comando `git config --global core.editor` tal como viste en [ch01-introduction]).

El editor mostrará el siguiente texto (este ejemplo corresponde a una pantalla de 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:
#       new file:   README
#       modified:   CONTRIBUTING.md
#
~
~
~
".git/COMMIT_EDITMSG" 9L, 283C
```

Puedes ver que el mensaje de confirmación por defecto contiene la última salida del comando `git status` comentada y una línea vacía encima de ella. Puedes eliminar estos comentarios y escribir tu mensaje de confirmación, o puedes dejarlos allí para ayudarte a recordar qué estás confirmando. (Para obtener una forma más explícita de recordar qué has modificado, puedes pasar la opción `-v` a `git commit`. Al hacerlo se incluirá en el editor el diff de tus cambios para que veas exactamente qué cambios estás confirmando). Cuando sales del editor, Git crea tu confirmación con tu mensaje (eliminando el texto comentado y el diff).

Otra alternativa es escribir el mensaje de confirmación directamente en el comando `commit` utilizando la opción -m:

```
$ git commit -m "Story 182: Fix benchmarks for speed"
[master 463dc4f] Story 182: Fix benchmarks for speed
 2 files changed, 2 insertions(+)
 create mode 100644 README
```

¡Has creado tu primera confirmación (o *commit*)! Puedes ver que la confirmación te devuelve una salida descriptiva: indica cuál rama has confirmado (`master`), que *checksum* SHA-1 tiene el *commit* (`463dc4f`), cuántos archivos han cambiado y estadísticas sobre las líneas añadidas y eliminadas en el *commit*.

Recuerda que la confirmación guarda una instantánea de tu área de preparación. Todo lo que no hayas preparado sigue allí modificado; puedes hacer una nueva confirmación para añadirlo a tu historial. Cada vez que realizas un *commit*, guardas una instantánea de tu proyecto la cual puedes usar para comparar o volver a ella luego.

Saltar el Área de Preparación

A pesar de que puede resultar muy útil para ajustar los *commits* tal como quieres, el área de preparación es a veces un paso más complejo de lo que necesitas para tu flujo de trabajo. Si quieres saltarte el área de preparación, Git te ofrece un atajo sencillo. Añadiendo la opción `-a` al comando `git commit` harás que Git prepare automáticamente todos los archivos rastreados antes de confirmarlos, ahorrándote el paso de `git add`:

```
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md

no changes added to commit (use "git add" and/or "git commit -a")
$ git commit -a -m 'added new benchmarks'
[master 83e38c7] added new benchmarks
 1 file changed, 5 insertions(+), 0 deletions(-)
```

Fíjate que en este caso no fue necesario ejecutar `git add` sobre el archivo `CONTRIBUTING.md` antes de confirmar.

Eliminar Archivos

Para eliminar archivos de Git, debes eliminarlos de tus archivos rastreados (o mejor dicho, eliminarlos del área de preparación) y luego confirmar. Para ello existe el comando `git rm`, que además elimina el archivo de tu directorio de trabajo de manera que no aparezca la próxima vez como un archivo no rastreado.

Si simplemente eliminas el archivo de tu directorio de trabajo, aparecerá en la sección “Changes not staged for commit” (esto es, *sin preparar*) en la salida de `git status`:

```
$ rm PROJECTS.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        deleted:    PROJECTS.md

no changes added to commit (use "git add" and/or "git commit -a")
```

Ahora, si ejecutas `git rm`, entonces se prepara la eliminación del archivo:

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

    deleted:    PROJECTS.md
```

Con la próxima confirmación, el archivo habrá desaparecido y no volverá a ser rastreado. Si modificaste el archivo y ya lo habías añadido al índice, tendrás que forzar su eliminación con la opción `-f`. Esta propiedad existe por seguridad, para prevenir que elimines accidentalmente datos que aun no han sido guardados como una instantánea y que por lo tanto no podrás recuperar luego con Git.

Otra cosa que puedas querer hacer es mantener el archivo en tu directorio de trabajo pero eliminarlo del área de preparación. En otras palabras, quisieras mantener el archivo en tu disco duro pero sin que Git lo siga rastreando. Esto puede ser particularmente útil si olvidaste añadir algo en tu archivo `.gitignore` y lo preparaste accidentalmente, algo como un gran archivo de trazas a un montón de archivos compilados `.a`. Para hacerlo, utiliza la opción `--cached`:

```
$ git rm --cached README
```

Al comando `git rm` puedes pasarle archivos, directorios y patrones glob. Lo que significa que puedes hacer cosas como

```
$ git rm log/\*.log
```

Fíjate en la barra invertida (`\`) antes del asterisco `*`. Esto es necesario porque Git hace su propia expansión de nombres de archivo, aparte de la expansión hecha por tu terminal. Este comando elimina todos los archivo que tengan la extensión `.log` dentro del directorio `log/`. O también puedes hacer algo como:

```
$ git rm \*~
```

Este comando elimina todos los archivos que acaben con `~`.

Cambiar el Nombre de los Archivos

Al contrario que muchos sistemas VCS, Git no rastrea explícitamente los cambios de nombre en archivos. Si renombras un archivo en Git, no se guardará ningún metadato que indique que renombraste el archivo. Sin embargo, Git es bastante listo como para detectar estos cambios luego que los has hecho - más adelante, veremos cómo se detecta el cambio de nombre.

Por esto, resulta confuso que Git tenga un comando `mv`. Si quieres renombrar un archivo en Git, puedes ejecutar algo como

```
$ git mv file_from file_to
```

y funcionará bien. De hecho, si ejecutas algo como eso y ves el estado, verás que Git lo considera como un renombramiento de archivo:

```
$ git mv README.md README
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed:    README.md -> README
```

Sin embargo, eso es equivalente a ejecutar algo como esto:

```
$ mv README.md README
$ git rm README.md
$ git add README
```

Git se da cuenta que es un renombramiento implícito, así que no importa si renombras el archivo de esa manera o a través del comando `mv`. La única diferencia real es que `mv` es un solo comando en vez de tres - existe por conveniencia. De hecho, puedes usar la herramienta que quieras para renombrar un archivo y luego realizar el proceso rm/add antes de confirmar.

prev | next
About this site
Patches, suggestions, and comments are welcome.
Git is a member of Software Freedom Conservancy