Introducción a Git

Diego D. Galizzi

August 26, 2010

Contents

Git - Sistema de control de versión

Control de versión consiste justamente en eso, en ir guardando todos los cambios que se hacen en un proyecto determinado. Entonces en cualquier momento podemos volver a una versión anterior, para lo que necesitemos.

Esto no es sólamente útil para programadores; a un músico, un diseñador, o cualquier tipo de artista que trabaje en la computadora le puede resultar muy útil dejar todos los cambios guardados. Si no te gustó algún cambio, podés volver en cualquier momento, a cualquier lugar, realizar comparaciones, etc. Cualquiera que maneje archivos que van cambiando con el tiempo, puede que le resulte muy útil utilizar este sistema.

El kernel de linux se mantuvo inicialmente a base de parches y paquetes de archivos. Uno realizaba algún cambio y mandaba el parche (archivo con las diferencias a la versión modificada) al mantenedor oficial. Obviamente este sistema para colaborar es muy incómodo, y muy difícil de mantener cuando hay muchos colaboradores y muchos cambios.

Un método muy *crudo* para realizar control de versión es crear distintas carpetas cada vez que se considera que se ha realizado un cambio significativo. Se podría nombrar cada carpeta con la fecha así se mantiene ordenado lo mejor posible.

El problema de este método es que se vuelve muy incómodo y desordenado. Además hay muchas posibilidades de error, nos podemos llegar a perder y modificar una versión equivocada, por ejemplo.

Lo primero que se realizó para tratar con esto son unos sistemas de control que van guardando en una base de datos cada versión, de forma local. Este programa va calculando las diferencias entre cada versión, entonces puede generar la versión que se desee.

El problema de este sistema es que no se puede trabajar en equipo, ya que la base de datos es local. Para resolver esto, se diseñaron los sistemas de control de versión **centralizados**.

Sistemas centralizados

Estos sistemas trabajan en un servidor particular, donde todas las personas con acceso al servidor pueden conectarse y colaborar (como Subversion, CVS y otros). Cada uno puede hacer sus propias modificaciones, ver las modificaciones de los demás, etc. De esta forma se lleva mucho mayor control a la hora de trabajar en equipo, cada uno sabe quién hizo qué, y cuando.

La desventaja de este sistema, es justamente que corra en un servidor en particular y que todo tenga que realizarse al servidor. Primero que por cada *commit* que se realice hay que conectarse a un servidor externo y la latencia se siente.

El segundo problema y más importante es que cuando el servidor se cae nadie puede contribuir hasta que vuelva a subir.

Otro problema es que nunca se debe guardar todo un proyecto en un único lugar, si bien esto se puede resolver con backups constantes, puede resultar muy pesado realizarlos.

Sistemas distribuídos

Con los sistemas distribuídos (como Git, Mercurial, Bazaar) cuando uno se conecta al servidor para obtener el proyecto, no sólo obtiene una copia instantánea de la versión actual, sino **todo** el repositorio. Es decir, uno puede trabajar localmente de forma completa, de esta forma se elimina latencia logrando trabajar más rápido.

Aún más importante, con este sistema **cada** colaborador tiene un backup completo del repositorio, y tampoco hay que esperar a que el servidor suba para ir realizando los *commit* necesarios.

Git

La comunidad desarrolladora de linux eventualmente necesitó un programa especialmente diseñado para el kernel (un proyecto inmenso con mucha colaboración), entonces decidieron crear su propia herramienta, que llamaron Git.

Los objetivos para Git eran (y siguen siendo):

- Velocidad
- Simple diseño
- Gran soporte para desarrollo no lineal (miles de ramas de desarrollo paralelas)
- Distribuído
- Capaz de manejar proyectos grandes

Git nació en el 2005 y desde ahí ha evolucionado de tal forma que se utiliza en muchos proyectos, es muy eficiente y veloz. Y realmente cumple con todos los objetivos planteados.

En Git la mayoría de las operaciones que uno realiza son locales, en general no se necesita información de otra computadora para poder trabajar con Git. Si uno quiere ver el historial, simplemente lo leemos de nuestro disco, sin necesidad de conectarse a la red. Podemos movernos a cualquier versión y cualquier rama de trabajo de forma prácticamente instantánea.

De esta forma se puede estar desconectado totalmente de redes externas y aún así seguir trabajando, apenas se pueda conectar al servidor se suben los cambios.

Los tres estados

Es importante saber cuáles son los tres estados principales de Git por los cuales se pasa constantemente en el ciclo de desarrollo.

Tenemos el directorio de trabajo, el índice (*staging* o *index* en inglés) y el repositorio. El directorio de trabajo es el proyecto en sí, en la versión y rama que estemos actualmente. Cada modificación que hagamos al proyecto se quedará en el directorio de trabajo, a menos que agreguemos dicha modificación al índice. El índice simplemente lleva la cuenta de que es lo que se agregará al repositorio. Finalmente se realiza un commit y todo lo marcado en el índice se agrega al repositorio de forma permanente.

Resumiendo, un ciclo común de trabajo con Git es:

1. Realizamos cambios en el proyecto.
2. Marcamos los cambios en el índice.
3. Hacemos un commit para realizar todos los cambios en el repositorio.
4. Ir a 1.

Sólo se agregarán al repositorio los archivos modificados, para el resto quedará un puntero/enlace a la última modificación del archivo.

Manos a la obra

La mayoría de las distribuciones de linux ofrecen alguna forma sencilla de instalar Git, así que no me voy a concentrar en esto. Para Windows existe el instalador msysgit.

Lo primero que conviene hacer es configurar unos datos globales, así todo lo que hacemos queda registrado correctamente:

```
git config --global user.name "Nombre del usuario"
git config --global user.email "Usuario@dom.com"
```

Una vez que tenemos git instalado y configurado (lo principal al menos) ya podemos arrancar. Nos dirigimos a un directorio vacío y hacemos:

```
$git init
Initialized empty Git repository in /home/diego/devel/blog/git/.git/
```

Si obtenemos el mensaje es porque todo está funcionando bien. El repositorio se inicializa en la carpeta .git.

Ahora creemos un par de archivos, por ejemplo un README.

```
$ touch README
```

Utilizando *git status* podemos ver cuál es el estado en que se encuentran nuestros cambios respecto a git:

```
$ git status
# On branch master
#
# Initial commit
#
# 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)
```

Obtenemos un mensaje con varias cosas nuevas. Primero dice "On branch master", esto significa que estamos en la *rama maestra*, con Git podemos crear distintas ramas paralelas de trabajo (más sobre esto después), la rama inicial es la llamada "master".

Luego en "Untracked files:" nos indica los archivos que Git no está siguiendo, o sea, en el repositorio no existe ninguna versión de dicho archivo.

Para agregar archivos al índice usamos *git add*, de esta forma:

```
$ git add README
```

Si ahora hacemos de nuevo:

```
$ git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#   new file:   README
#
```

Vemos que la sección de "Untracked files" no está mas, y ahora está la sección "Changes to be commited", donde está el archivo README que agregamos recién. Esto significa que si ahora hacemos un commit, este archivo se agregará al repositorio. Para hacer un commit utilizamos *git commit*:

```
$ git commit -m "Primer commit"
[master (root-commit) 8b6bc3c] Primer commit
 0 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 README
```

El parámetro -m es para indicar un mensaje que quedará guardado en el commit, se recomienda utilizar una descripción adecuada del cambio realizado. Si no utilizamos -m Git nos abrirá un editor de texto para hacerlo. Se puede modificar el editor preferido de la siguiente forma:

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

Si ahora hacemos de nuevo:

```
$ git status
# On branch master
nothing to commit (working directory clean)
```

Nos dice que no hay nada que hacer, está todo actualizado. Modifiquemos el README y agreguemos un archivo nuevo:

```
$ echo "read me" >> README
$ touch LICENSE
```

Nuevamente:

```
$ git status
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#   modified:   README
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#   LICENSE
no changes added to commit (use "git add" and/or "git commit -a")
```

Ahora git nos marca que el archivo README ha sido modificado pero que no está en el índice, y que hay un archivo LICENSE nuevo.

En el git add también se pueden usar comodines, así que podemos agregar todo de una:

```
git add *
```

Luego hacemos commit, y continuamos con el ciclo básico de desarrollo. De la misma manera, si removemos un archivo, git lo mostrará como tal:

```
$ rm LICENSE
$ git status
# On branch master
# Changed but not updated:
#   (use "git add/rm <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#   deleted:    LICENSE
#
no changes added to commit (use "git add" and/or "git commit -a")
```

Alternativamente podemos hacer el commit y al mismo tiempo mandar al índice todo lo modificado usando *git commit -a*.

Branches

Una de las grandes ventajas de Git es el manejo de las branches, son muy livianas (41 bytes, ya que en realidad son punteros), cómodas y fáciles de usar. Git está diseñado específicamente para lograr un buen trabajo utilizando branches.

En general es cómodo trabajar en ramas paralelas. La rama master es adecuada para dejar con la última versión estable o que se considere importante.

Trabajando así, el proceso cambia un poco:

1. Creamos una nueva rama y nos movemos a ella.
2. Trabajamos en esta nueva rama con lo que tengamos planeado para la próxima versión.
3. Combinamos esta rama con la estable.
4. Ir a 1.

La ventaja de trabajar de esta manera es que llevamos un mejor registro de qué parte es estable y cuál de desarrollo. También se puede hacer una rama que llamamos estable y la master la dejamos de desarrollo.

Puede haber personas interesadas en obtener distintas versiones del programa, algunas prefieren la última versión (aún así sea inestable), y otras prefieren tener la versión estable. Separando en branches estas características resulta muy cómodo para los usuarios que siguen el repositorio, cada uno sigue la que le interesa.

Supongamos que estamos en nuestra rama de desarrollo y se reporta un bug muy importante en la rama estable, lo que podemos hacer es volver a la estable, crear una nueva rama para arreglar el bug y finalmente combinar esta rama con la estable. Después podemos volver a la rama de desarrollo, y *mezclar* los cambios que hicimos en la estable así tenemos el bug corregido en todos lados.

Otro ejemplo que se da mucho es el de ramas experimentales, en estas ramas empezamos desarrollo con características que quizás no estaban previstas de antemano y que se quiere experimentar. Es conveniente no mezclar este tipo de desarrollo con las otras ramas, entonces si llega a no funcionar o no resultar como uno esperaba se puede abandonar con la menor pérdida posible.

Branches en Git

Para ver las branches usamos *git branch* sin más argumentos, obtenemos

```
$ git branch
* master
```

Vemos que la única branch es la master, para crear una nueva usamos *git branch nombre* donde *nombre* es el nombre que le queremos dar a la branch, por ejemplo:

```
$ git branch fix
$ git branch
  fix
* master
```

Primero con *git branch fix* creamos una nueva rama llamada *fix*, luego las mostramos y vemos que aparecen las dos (la master y la fix). La master lleva un asterisco adelante debido a que es en la que estamos trabajando actualmente. Para movernos a la nueva rama usamos *git checkout branch*, de la siguiente manera:

```
$ git checkout fix
Switched to branch 'fix'
```

Además del mensaje podemos verificar que estamos en la nueva rama:

```
$ git branch
* fix
  master
```

Como lo marca el asterisco. Alternativamente se puede crear una nueva rama y moverse a ella en un solo comando, haciendo *git checkout -b rama*.

Ahora podemos realizar los cambios necesarios, hacer commit, y luego unir la rama master con la rama fix de la siguiente manera:

```
$ touch FIX
$ git add FIX
$ git commit -m "Fixed."
[fix 7868a0b] Fixed
 0 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 FIX
$ git checkout master
Switched to branch 'master'
$ git merge fix
Updating 625f3b4..7868a0b
Fast forward
 0 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 FIX
```

Utilizamos *git merge branch* para unir con otra rama, vemos que luego del merge Git nos dice "Fast forward", esto significa que Git simplemente movió el puntero de la master hacia adelante. Después de crear en la rama fix nunca volvimos a la rama master, por lo tanto la rama fix estaba completamente por delante de la master, sin divergir.

Notar que la rama fix no se eliminó:

```
$ git branch
  fix
* master
```

Sin embargo en las dos ramas existe el mismo contenido. Ahora ya podríamos borrar la rama fix, porque no la necesitamos, lo hacemos con *git branch -d branch*:

```
$ git branch -d fix
Deleted branch fix (was 7868a0b).
```

Resolución de conflictos

El merge no siempre sale limpio, puede haber conflictos que git no sabe como resolver, en este caso somos nosotros que tenemos que resolverlo.

Supongamos que tenemos un repositorio vacío, y hacemos lo siguiente

```
$ touch "README"
$ git add .
$ git commit -m "README"
$ git checkout -b otra
$ echo "read me 1" >> README
$ git commit -a -m "Read me 1"
$ git checkout master
$ echo "read me 2" >> README
$ git commit -a -m "Read me 2"
$ git merge otra
Auto-merging README
CONFLICT (content): Merge conflict in README
Automatic merge failed; fix conflicts and then commit the result.
$
```

Rápidamente, lo que hice acá fue: Crear un archivo README, agregarlo y hacer el commit (dentro de la rama master). Después creé una nueva rama y me moví a ella (*git checkout -b otra*), dentro de esta rama modifiqué el archivo readme con el contenido "read me 1".

Luego volví a la rama master y modifiqué el archivo (en este momento vacío, ya que el cambio lo realicé en la otra rama) con otro contenido: "read me 2". En este momento intenté hacer el merge, pero git me dice que hay conflictos, ya que cada archivo tiene contenido distintos y es en la misma línea del archivo (además, por el contexto, git no puede determinar como solucionarlo).

Git nos dice que resolvamos el conflicto y agreguemos el archivo nuevamente. Podemos hacer git status en cualquier momento para ver qué archivos están en conflicto:

```
$ git status
# On branch master
# Unmerged paths:
#   (use "git add/rm <file>..." as appropriate to mark resolution)
#
#   both modified:      README
#
no changes added to commit (use "git add" and/or "git commit -a")
```

Si abrimos el archivo README vemos que Git agregó unos marcadores estándar para remarcar el conflicto:

```
<<<<<<< HEAD
read me 2
=======
read me 1
>>>>>>> otra
```

La primer parte antes del "=======" es el llamado "HEAD", HEAD es un puntero que apunta a la rama en la que estamos actualmente, en este caso la master (recordar que las ramas son a la vez punteros). La siguiente parte es el contenido del archivo en la rama "otra".

Para solucionar el conflicto debemos reemplazar todo el bloque en cuestión (incluyendo los marcadores) por la resolución, por ejemplo, podríamos reemplazar todo por:

```
read me 1
read me 2
```

Luego tenemos que agregar el archivo para que git termine el merge, utilizando *git add*:

```
$ git add README
$ git status
# On branch master
# Changes to be committed:
#
#   modified:   README
#
$ git commit -m "Merge con otra"
[master 355c14e] Merge con otra
```

Finalmente hice el commit para terminar de guardar los cambios en el repositorio. Para resolver los conflictos se pueden utilizar herramientas especializadas con el comando *git mergetool* (herramientas válidas son kdiff3, tkdiff, meld, xxdiff, emerge, vimdiff, gvimdiff, ecmerge, diffuse, tortoisemerge, opendiff, p4merge y araxis)

GitHub

Hasta ahora trabajamos sólo sobre el repositorio de forma local, pero se puede trabajar en un servidor. Uno puede montar su propio servidor, pero acá voy a mostrar como usar un servidor gratuito para subir repositorios públicos en Git.

GitHub es el servicio de hosting de repositorios de Git más utilizado actualmente. Una característica muy importante de GitHub es que nos permite hacer *forks* (una copia de un repositorio y automáticamente crear otro en GitHub) de manera muy sencilla, por medio de un simple click. La facilidad de realizar forks habilita un nuevo ciclo de trabajo colaborativo que resulta muy cómodo.

Supongamos que hay un programa al que queremos colaborar y está alojado en GitHub, podemos hacerle un fork y de esta forma creamos nuestro propio repositorio en Github. Bajamos este repositorio a nuestra computadora para trabajarlo localmente. Creamos una nueva branch y realizamos el cambio que queremos hacer y subimos estos cambios al fork que hicimos en GitHub.

Finalmente GitHub ofrece otra herramienta muy útil, el *pull request*, vamos al repositorio oficial y le damos click en *pull request*, de esta forma le estamos pidiendo al desarrollador oficial que haga un *pull*, esto es, que combine nuestro fork con su repositorio. El desarrollador luego puede hacer un pull local para probar nuestros cambios, y si le gusta lo agrega al repositorio oficial.

Branches remotas

Las branches remotas son ramas que marcan el estado de las ramas locales en los repositorios remotos. Su nombre completo tiene la forma (remote)/(branch), donde *remote* es un nombre especial de la rama remota y *branch* es en que rama se encuentra. Estas ramas las movemos con comandos específicos que requieren conectarse al servidor remoto.

Para crear una rama remota utilizamos *git remote add remote server*. Donde remote es el nombre de la rama (sólo la parte (remote), la parte (branch) la deduce automáticamente de la rama actual) y *server* es el repositorio remoto de git.

Es muy común nombrar el repositorio oficial con el nombre *origin* y así es como lo nombra git cuando hacemos *git clone* (ver más abajo).

Si por ejemplo hacemos un repositorio nuevo en GitHub toma la forma:

```
git@github.com:(nombre de usuario)/(nombre del repositorio).git
```

Entonces, una vez que creamos nuestro repositorio en github, podemos agregar una rama remota al servidor de la siguiente manera:

```
$ git remote add origin git@github.com:(nombre de usuario)/(nombre del repositorio).git
```

De esta forma creamos una nueva rama remota llamada *origin* (podríamos ponerle cualquier nombre). Podemos ver estas ramas ejecutando *git remote* sin argumentos:

```
$ git remote
origin
```

Ahora podemos usar *git push remote branch* para guardar los cambios de nuestro repositorio local a GitHub, de la siguiente forma:

```
$ git push origin master
```

Estamos haciendo un *push* de los cambios, en la rama master, al repositorio indicado por la rama remota *origin*.

Nota: Cuando creas un nuevo repositorio en GitHub te muestra una receta para hacer lo que mostré acá, pero no explica bien que es cada cosa.

Si hay alguien más trabajando en el mismo repositorio remoto, puede pasar que al hacer push haya conflictos, esto sucede porque alguien pudo haber actualizado el repositorio remoto y nosotros veníamos trabajando con contenido anterior. Para actualizar el contenido localmente utilizamos *git fetch remote*, donde *remote* es el nombre de la rama remota, por ejemplo:

```
$ git fetch origin
```

Nos va a actualizar la rama remota para estar a la misma altura que el servidor. Esta operación nunca modifica ninguna de nuestras ramas y es totalmente seguro, luego podría ser necesario hacer un merge para unir el trabajo agregado al servidor con el nuestro local, y finalmente un push para subir todo.

Alternativamente se puede utilizar *git pull (remote) (branch)* para hacer un fetch y luego un merge de forma conjunta, por ejemplo:

```
$ git pull origin master
```

Es equivalente a:

```
$ git fetch origin
$ git merge origin/master
```

Es decir, actualizamos las referencias del servidor *origin* y después hacemos un merge de la rama actual con la rama master en el servidor remoto.

Git clone

Se puede utilizar el comando *git clone* para clonar repositorios. Con git clone se clona un repositorio a un directorio local nuevo. Se utiliza generalmente la primera vez que descargamos un repositorio, ya que después podemos utilizar *git fetch* o *git pull* para descargar los cambios.

La forma más común de usar *git clone* es *git clone (repositorio) (directorio)*, por ejemplo, si queremos descargar un repositorio que se encuentra en git://repositorio.com/repo.git/ en el directorio *repo* hacemos:

```
git clone git://repositorio.com/repo.git/ repo
```

Git y SVN

Git se puede combinar con SVN (Subversion) de manera muy sencilla, ya que trae una herramienta oficial para esto. Esta herramienta resulta útil para descargar repositorios de SVN pero trabajar con git localmente, luego podemos subirlo al SVN por medio de git, y git se encarga de todo.

Esta combinación nos permite además una forma muy sencilla de migrar de SVN a Git.

No me quiero extender mucho con esto, sólo remarcar lo básico. Los comandos más comunes para trabajar con git svn son:

Nota: Se necesita tener SVN instalado.

Crear un repositorio de git a partir de uno SVN:

```
$ git svn clone http://svn-repositorio.com directorio_local
```

Obtener lo último del repositorio SVN:

```
$ git svn fetch
$ git svn rebase
```

Subir los cambios al servidor SVN:

```
$ git svn dcommit
```

Los commits locales se realizan de la misma forma de siempre.

Conclusión

Git suele ser un poco más complicado de entender y aprender que otros sistemas de control de versión, pero sin dudas tiene sus ventajas. Muchas cosas se van aprendiendo usandolo y leyendo la documentación a medida que uno se enfrenta a algún problema, siempre hay alguna solución.

Hay **mucho** más de git de lo que puse acá, pero con esto es suficiente para sentir comodidad al trabajar con git y para el uso cotidiano que un desarrollador promedio puede darle. Una forma de aprender realmente git es aprendiendo como funciona internamente, de esta forma uno logra entender todos los comandos y algunos *trucos* útiles.

Lecturas adicionales

Como lecturas adicionales para aprender más sobre git recomiendo:

Pro Git: Un libro de libre distribución muy completo sobre Git, al final de la página pueden encontrar traducciones, incluyendo una traducción (incompleta en el momento de escribir esto) al español. Alternativamente se puede comprar una versión impresa.

Documentación oficial: Obviamente también recomiendo la documentación oficial, especialmente para aprender comandos específicos y utilizarla como referencia.

gitref: Esta página incluye un tutorial similar al que escribí yo aquí pero más completo.

gitready: Recetas y *tips* de git.

Tech talk: Charla técnica de git por Linus Torvalds. Alrededor de una hora de duración.

Tech talk: Charla técnica por Randal Schwartz. Según Randal esta es una charla sobre "¿Qué es git?", mientras que la charla de Linus es sobre "¿Qué **no** es git?". Alrededor de una hora de duración.