Guía para el uso de Git en Linux

Algoritmos y Programación I

Julián García Delfino

March 26, 2019

Contents

1 Introducción a Git

1.1 ¿Qué es Git?

Git Es un software de control de versiones que se usa para poder hacer un correcto seguimiento al desarrollo de programas que tienen archivos extensos que contienen sus códigos.

Y entonces, ¿Cómo funciona?

1.1.1 Definiciones que nos pueden ayudar

Repositorio En sistemas de revisión de versiones, como Git, un repositorio es una estructura de datos que guarda **metadata** de un conjunto de archivos.

Metadata Es informacion sobre nuestros archivos. *"Data that provides information about other data"*

1.1.2 Idea general del funcionamiento

Con la información que tenemos hasta ahora, podríamos esbozar, en el aire, una idea de cómo funciona Git.
Al crear un repositorio, estamos habilitando al sistema de revisión de versiones a que guarde cierta información sobre los archivos que nosotros pongamos en ese directorio. Es decir, si tenemos nuestro código con extensión, por ejemplo, **.c** en una carpeta que a la vez es un repo de Git, éste puede, gracias a la metadata que almacena, llevar la cuenta de los cambios que se generan entre versiones del programa. Esto podría ser muy conveniente si estamos trabajando en algo muy complejo, o incluso sólamente extenso.

1.2 Funcionamiento de Git

Entonces, ya generamos una idea sobre qué es lo que hace Git con nuestros datos, pero puede ser que todavía sea muy vaga y abstracta.

Básicamente, nuestra información se moverá entre 3 etapas:

- Working Directory
- Staging Area
- Repository

1.2.1 Working Directory

El **Working Directory** es, básicamente, donde tendremos almacenados los archivos sobre los que trabajaremos y donde luego, también, crearemos el repositorio de Git.
Por ejemplo:

```
$ sudo mkdir proyecto1 /*Creo la carpeta donde almacenaré los archivos*/
$ cd proyecto1 /*Me muevo dentro de esa carpeta*/
$ git init
Initialized empty Git repository in /direcciónPrevia/proyecto1/.git
```

Ahora **proyecto1** es mi **Working Directory**.

1.2.2 Staging Area

El **Staging Area** es la etapa donde vamos a preparar el **commit**, o por ahora, la **entrega**.

Como veremos mejor mas adelante, no basta con realizar cambios a los archivos, sino que hay que *"agregarlos"* al **Staging Area**.
Imagínense que estamos haciendo el trabajo práctico de la materia, pero, antes de seguir avanzando queremos ir llevando la cuenta de los cambios importantes que le hacemos a nuestros archivos. Entonces, tenemos creada la carpeta, el *Working Directory*, y dentro de ella inicializamos el repo de Git. Ya hicimos cambios y queremos asentarlos.
Git, a esta altura, sabe todas las modificaciones que se fueron haciendo en el tiempo, desde la creación del repositorio, pero hasta que no las agregamos al **Staging Area**, éste no podrá hacer nada con esta información.
Entonces, agregamos todos los archivos que querramos entregar y formamos un *paquete* con estos. El **Staging Area** se encarga de eso, de ser el paquete que entregaremos con toda la información que necesita Git para seguir el rasto.

1.2.3 Repository

El **Repositorio** es donde Git almacenará toda la metadata que necesita para generar el registro de control de versiones.

Entonces, tenemos el paquete armado en el *Staging Area* y queremos mandarlo al **repositorio**, basta hacer el *commit* y listo. Git tendrá toda la información que necesita. Pero, ¿Cómo sabemos nosotros qué cosas se cambiaron concretamente? Bueno, con cada *commit* se debe poner un mensaje que especifique de forma clara qué cosas se cambiaron.

Por ejemplo:

```
"Implementación de la función de suma"
                    ^
                    |
                 Commit
    |Paquete| -------> |Repositorio|
```

Panorama General

Ahora, ya tenemos todas las piezas del rompecabezas que es el funcionamiento de Git.
Si lo pusiésemos todo junto nos quedaría algo así:

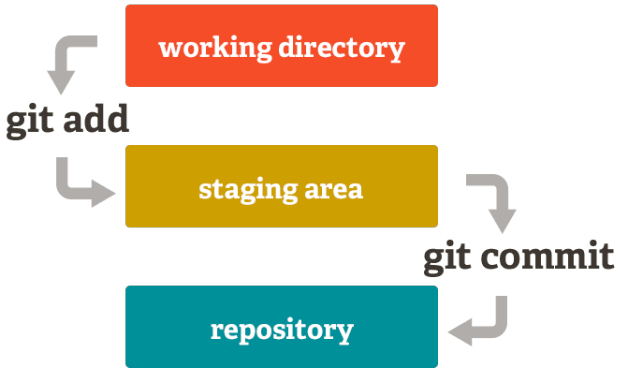

Figure 1: Esquema del funcionamiento de Git.

2 Instalación de Git

En el cmd (Ctrl + Alt + t):

```
$ sudo apt install git-all
```

Obs: Este comando funciona para distros basados en Debian (Como Ubuntu).

2.1 Configuración Opcional

Hay algunas cosas más que podemos configurar de Git que podrían, en algún momento, sernos útiles.

```
$ git config --global user.name "Tony Stank"
```

Ésto lo que hace es, configurar un estilo de sello que irá en cada paquete que se commitée a Git.

```
$ git config --global user.email "direcciónDeCorreo@Algoritmos1.com"
```

Acá, a ese sello, le agregamos nuestro mail. Puede ser que no les parezca algo que pueden llegar a usar ahora, y tienen razón. Éste es, capaz, un aspecto de Git que puede ser muy útil en proyectos más grandes, con más personas. Si alguno de sus integrantes necesita contactarse con la persona que hizo un commit específico al código, puede hacerlo directamente a esa dirección.

3 Crear un repositorio

Primero, debemos movernos a la posición dentro de la PC donde querramos inicializar el repo, es decir, donde vamos a guardar nuestro proyecto (Comandos cd y mkdir).

Una vez ahí:

```
$ mkdir prueba
$ cd prueba
$ git init
Initialized empty Git repository in /Dirección/.git/
```

3.1 Archivo README

Es buena costumbre agregar un archivo README para informar exactamente para qué será utilizado el repo:

```
$ nano README.txt
```

Como sabemos, nano es un editor de texto que viene integrado con la consola de linux.
Para confirmar que guardaron el archivo correctamente podrían usar el comando:

```
$ ls
README.txt
```

También, existe el comando `$ git status` que nos mostrará los cambios hechos en los archivos que hemos indicado que debe seguir. En el siguiente ejemplo, Git nos avisará que hay un archivo, pero no le esta haciendo el seguimiento.

```
$ git status
On branch master

No commits yet

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

        README.txt

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

4 Add y Commit

Un **commit** es como un registro de actividades dentro del repo, es decir, una *entrega*. Cada vez que se haga un **commit**, Git registrará tanto la adición como la modificación de archivos y llevará la cuenta. Ésto es lo que nos facilitará el control de versiones en un futuro.

Git detectará los cambios hechos en el repositorio, pero a menos que usemos el comando `$ git add <filename>`, éste no hará nada con ellos. Es decir:

```
$ git add README.txt
$ git status
On branch master

No commits yet

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

        new file:   README.txt
```

Ahora Git ya conoce nuestro archivo y debemos hacer un **commit**:

```
$ git commit -m "Mensaje descriptivo sobre las modificaciones y adiciones"
```

Por ejemplo:

```
$ git commit -m "Primera subida - README"
```

Si hemos hecho cambios en nuestro archivo y no hemos hecho un **commit** porque decidimos que queremos revertirlos, podemos usar:

```
$ git checkout -- <filename>
$ git checkout -- .
```

Ésto revertirá los cambios por archivo, individualmente, o a todos a la vez.

5 Branches y Merge

Supongamos que ahora queremos agregar alguna función nueva al programa pero no queremos, todavía, generar cambios en el proyecto principal. Para esto usamos un **Branch●**.

Para crear un Branch nuevo:

```
$ git checkout -b <branchName>
```

¿Por qué **checkout**? Una vez creado el **Branch**, como queremos trabajar en él, Git nos saca del master branch y nos mueve a esta rama recien creada.
Para chequear los branches disponibles podemos usar:

```
$ git branch
```

Siguiendo con nuestor ejemplo:

```
$ git checkout -b "branch1"
Switched to a new branch 'branch1'
```

```
$ git branch
* branch1
master
```

Para cambiar entre repositorios existentes:

```
$ git checkout "existingBranchName"
```

Los cambios hechos en **branch1** no afectarán a los commits en el branch principal **master**. Una vez finalizado el desarrollo en la rama secundaria, se podría mergear el contenido de ambas en la principal:

```
$ git merge branch1
```

6 Extra

6.1 Comandos Extra

6.1.1 Help

Para ver la documnetación de Git podemos usar:

```
$ git help
```

Y también, individualmente por argumento:

```
$ git argumentName --help
$ git add --help /*Por ejemplo*/
```

6.1.2 Stashing

El **Stash** es una herramienta muy útil que nos permite *almacenar* o justamente *stash* en una *repisa* o *shelf*, temporalmente, los cambios hechos en nuestros archivos por branch.

En otras palabras, imaginen que estamos trabajando en un *branch != master* que estamos usando para la implementación de una función específica. Por alguna razón, tenemos que cambiar de branch para retomar en otra sección del programa pero no terminamos la implementación en la que estábamos. Una parte de código sin terminar puede traer muchos dolores de cabeza, entonces, es conveniente deshacerse de los cambios que hicimos sin necesariamente perderlos, así en un futuro lo terminamos.

Acá es donde entra el stashing. Ilustraré los comandos con un ejemplo simple.

Supongamos que estamos implementando un programa que tiene dos funciones, una de suma y una de resta. Como buena costumbre, crearemos un branch para cada función para poder trabajar tranquilamente en ellas por separado.

Estando en branch *Suma*, quiero pasar al branch *Resta* pero no quiero perder los cambios en la función de suma:

```
$ git diff /*Nos muestra los cambios que hicimos actualmente*/
$ git stash save "Implementación de la función suma"

$git stash list /*Lista de stashes almacenados en este branch*/
stash{0}: On suma: Implementación de la función suma
```

Nuestros cambios están guardados en un estante virtual, y nosotros podemos seguir trabajando.
Cuando sea que querramos volver a la función de suma, y ver los cambios que guardamos, tendríamos que hacer:

```
$ git stash apply stash{0}
$ git stash list
stash{0}: On suma: Implementación de la función suma
```

Como vemos, el stash sigue en la lista, y esto es porque con el comando *apply* los cambios se revirtieron, pero igual se conserva el stash. Si no se especifica que stash usar, usará el último.
Tambień se podría usar:

```
$ git stash pop
$ git stash list
```

Ésta vez, los cambios también se restauran y el stash ha sido eliminado de la lista. Siempre usa el último stash.

Dato: Los stashes existen en todos los branches a la vez!
Si quisiésemos borrar los stashes:

```
$ git stash drop stash{0}
$ git stash clear /*Borra todos los stashes*/
```

6.2 Información de Git - Bibliografía

- Un about de Git
- Git Internals Explained Like I'm Five
- ¿Qué es un repositorio?
- ¿Qué es la metadata?
- ¿Qué es un sistema de control de versiones?

7 Repositorios con GitHub

7.1 ¿Git grupal?

Aprendimos el funcionamiento basico de Git como herramienta para proyectos, pero, **¿Qué pasa si quiero trabajar en grupo?**

Para eso hay servicio como **GitHub** y **GitLab**, entre otros. Al usar éstos, podemos llevar registro de todos los cambios y modificaciones de un proyecto grupal. **Nosotros vamos a proceder en la guía usando GitHub**.

7.2 Cuenta de GitHub

Primero nos creamos una cuenta, loggeamos y una vez en la pagina principal de GitHub.com cliqueamos en el boton de **New**

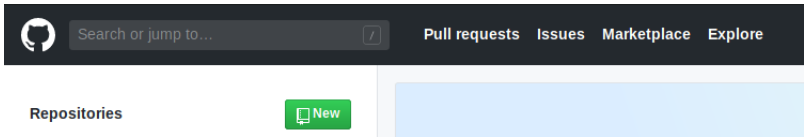

Figure 2: Captura del botón new en GitHub.

Lo nombramos y listo!
Ahora hay que linkear el repositorio local y el que acabamos de crear.

Desde la consola:

```
$ git remote add origin https://github.com/user/Prueba.git
$ git push --all origin

Username for 'https://github.com': userName
Password for 'https://juligarcia@github.com': userPassword
Counting objects: 11, done.
Delta compression using up to 4 threads.
Compressing objects: 100\% (7/7), done.
Writing objects: 100\% (11/11), 832 bytes | 416.00 KiB/s, done.
Total 11 (delta 3), reused 0 (delta 0)
remote: Resolving deltas: 100\% (3/3), done.
To https://github.com/juligarcia/Prueba.git
* {[new branch]}      master -> master}
```

Ya tenemos linkeado nuestro repo local con el que está en GitHub.
Si quisiésemos pushear un sólo branch deberíamos usar:

```
$ git push origin branchName
```

8 Fork y Merge

Esta herramienta es específica de servicios como GitHub.
Antes, habíamos hablado de como proceder si se quisiese desarrollar una parte del código por separado, pero, ¿qué haríamos si necesitáramos trabajar sobre el código principal? Es decir, cómo haíamos para modificar a éste, visible sólo para nosotros, y luego reunir todos las modificaciones en el repositorio principal.

Para eso estan los **Forks**. Ésta herramienta genera un clón del repositorio principal para que podamos modificar a gusto, sin, a la vez, cambiar el original.

Desde GitHub debemos ir al repo que queremos clonar, y ciquear en **Fork**

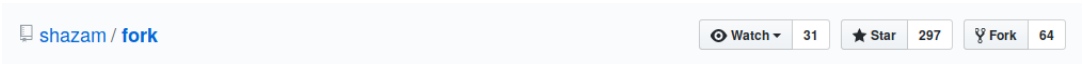

Figure 3: Captura del botón fork en GitHub.

Ahora, deberíamos tener dentro de nuestro repositorio personal, un fork con el mismo nombre que el original. Nos falta tener los archivos del repo localmente.
Dentro del repositorio original, copiamos el URL para clonarlo.
Volviendo a la consola de comandos:

```
$ git clone https://github.com/username/forkName

Cloning into 'Shazam'...
remote: Enumerating objects: 16, done.
remote: Total 16 (delta 0), reused 0 (delta 0), pack-reused 16
Unpacking objects: 100\% (16/16), done.
```

Podemos sincronizar nuestro fork con el repositorio principal, para que Git baje los cambios hechos a éste. Usando la direccion URL de la pagina del repo que forkeamos `https://github.com/originalUsername/f` abrimos la terminal y navegamos a la carpeta:

```
$ git remote -v
        origin    https://github.com/username/originalName.git (fetch)
        origin    https://github.com/username/originalName.git (push)

$ git remote add upstream https://github.com/originalUsername/forkName
$ git remote -v
        origin    https://github.com/username/originalName.git (fetch)
        origin    https://github.com/username/originalName.git (push)
        upstream    https://github.com/username/originalName.git (fetch)
        upstream    https://github.com/octocat/originalName.git (push)
```

Ahora sincronizamos nuestro fork con el repo original:

```
$ git fetch upstream
$ git merge upstream/master
Already up to date.
```

9 Pull Request

Imaginemos que acabamos de terminar el desarrollo de un branch en el que estabamos trabajando, implementando un *feature* para nuestro programa. Queremos hacer un *merge* con el master branch pero primero debemos discutirlo con el resto de los desarrolladores.
Para eso sirve un **Pull Request**, notifica a todos los involucrados en el proyecto y crea un entorno para poder discutir los cambios antes de hacer el merge.

Figure 4: Botón de Pull Request.

10 Control de versiones Local

Teniendo todas las herramientas para manejarnos con Git, nos queda implementarlas y hacer un buen control de versiones de nuestro programa.

Una primera herramienta que podemos usar con este fin es:

```
$ git log
$ git log --oneline /*Version del comando mas simple*/
$ git log --oneline -n /*Muestra los últimos n commits*/
$ git log --stat /*Muestra más información sobre los cambios*/
```

Este comando nos mostrará todos los commits hechos, su mensaje e información relevante, como fecha, hora, nombre y mail del autor.

Previamente, en la sección de *Add y Commit* vimos como deshacer cambios que no han sido pusheados al repositorio. Ahora veremos como hacerlo en el caso contrario.

Imaginense que hicimos varias entregas a nuestro repositorio, pero, como fuimos descuidados, nuestros cambios rompieron el programa y no funciona. Decidimos que estas modificaciones no son substanciales y que preferimos revertir el estado hasta la última versión que funcionaba.
Haremos lo siguiente:

```
$ git log --oneline -3
73abf7d cambio 3
308e338 cambio 2
dacacaf cambio 1
$ git revert dacacaf /*Hace el commit automáticamente*/
$ git revert -n dacacaf /*Modifica el archivo sin hacer commit*/
```

Esa "palabra" que aparece ates del mensaje de commit, por ejemplo "dacacaf", es el ID de éste.

A veces, cuando se usa este comando, puede haber conflicto con los cambios que se quieren realizar y Git agregará indicadores en el código para señalar donde es que encontro conflicto.
Un comando para hacerlo de otra forma sería:

```
$ git reset --hard dacacaf
```

Ésto revertirá el archivo completamente hasta ese commit descartando así todos los commits posteriores.