



GitHub

Empezando - Una breve historia de Git

Como muchas de las grandes cosas en esta vida, Git comenzó con un poco de destrucción creativa y encendida polémica. El núcleo de Linux es un proyecto de software de código abierto con un alcance bastante grande. Durante la mayor parte del mantenimiento del núcleo de Linux (1991-2002), los cambios en el software se pasaron en forma de parches y archivos. En 2002, el proyecto del núcleo de Linux empezó a usar un DVCS propietario llamado BitKeeper.

En 2005, la relación entre la comunidad que desarrollaba el núcleo de Linux y la compañía que desarrollaba BitKeeper se vino abajo, y la herramienta dejó de ser ofrecida gratuitamente. Esto impulsó a la comunidad de desarrollo de Linux (y en particular a Linus Torvalds, el creador de Linux) a desarrollar su propia herramienta basada en algunas de las lecciones que aprendieron durante el uso de BitKeeper. Algunos de los objetivos del nuevo sistema fueron los siguientes:

- Velocidad
- Diseño sencillo
- Fuerte apoyo al desarrollo no lineal (miles de ramas paralelas)
- Completamente distribuido
- Capaz de manejar grandes proyectos (como el núcleo de Linux) de manera eficiente (velocidad y tamaño de los datos)

Desde su nacimiento en 2005, Git ha evolucionado y madurado para ser fácil de usar y aún conservar estas cualidades iniciales. Es tremendamente rápido, muy eficiente con grandes proyectos, y tiene un increíble sistema de ramificación (branching) para desarrollo no lineal



Empezando - Fundamentos de Git

Fundamentos de Git

Entonces, ¿qué es Git en pocas palabras? Es muy importante asimilar esta sección, porque si entiendes lo que es Git y los fundamentos de cómo funciona, probablemente te sea mucho más fácil usar Git de manera eficaz. A medida que aprendas Git, intenta olvidar todo lo que puedas saber sobre otros VCSs, como Subversion y Perforce; hacerlo te ayudará a evitar confusiones sutiles a la hora de utilizar la herramienta. Git almacena y modela la información de forma muy diferente a esos otros sistemas, a pesar de que su interfaz sea bastante similar; comprender esas diferencias evitará que te confundas a la hora de usarlo.

¿Qué es GitHub?

Git es un sistema de control de versiones distribuida que se origina a partir del desarrollo del kernel de Linux y es usado por muchos proyectos populares Open Source como ser Android o Eclipse, así como tantos otros proyectos comerciales. Entonces, la principal diferencia entre Git y cualquier otro sistema de control de versiones es cómo Git modela sus datos. Conceptualmente, la mayoría de los demás sistemas almacenan la información como una lista de cambios en los archivos. Estos sistemas modelan la información que almacenan como un conjunto de archivos y las modificaciones hechas sobre cada uno de ellos a lo largo del tiempo. En cambio, Git modela sus datos más como un conjunto de instantáneas de un mini sistema de archivos. Cada vez que confirmas un cambio, o guardas el estado de tu proyecto en Git, él básicamente hace una foto del estado de todos tus archivos en ese momento, y guarda una referencia a esa instantánea. Para ser eficiente, si los archivos no se han modificado, Git no almacena el archivo de nuevo —sólo un enlace al archivo anterior idéntico que ya tiene almacenado.

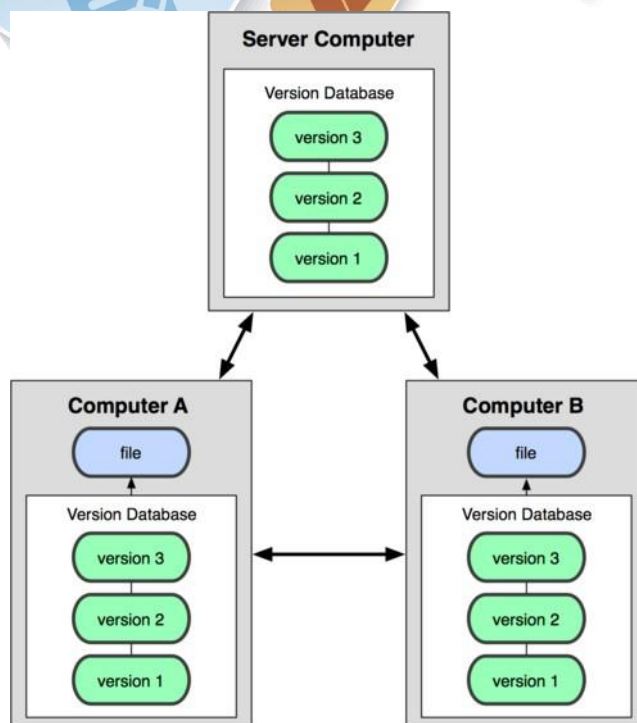


Ilustración 1

¿Para qué sirve?

GitHub aloja tu repositorio de código y te brinda herramientas muy útiles para el trabajo en equipo, dentro de un proyecto, además de eso, puedes contribuir a mejorar el software de los demás. Para poder alcanzar esta meta, GitHub provee de funcionalidades para hacer un fork y solicitar pulls.

Realizar un fork es simplemente clonar un repositorio ajeno (genera una copia en tu cuenta), para eliminar algún bug o modificar cosas de él. Una vez realizadas tus modificaciones puedes enviar un pull al dueño del proyecto. Éste podrá analizar los cambios que has realizado fácilmente, y si considera interesante tu contribución, adjuntarlo con el repositorio original.

¿Qué herramientas proporciona?

En la actualidad, GitHub es mucho más que un servicio de alojamiento de código. Además de éste, se ofrecen varias herramientas útiles para el trabajo en equipo. Entre ellas, caben destacar:

- Una wiki para el mantenimiento de las distintas versiones de las páginas.

- Un sistema de seguimiento de problemas que permiten a los miembros de tu equipo detallar un problema con tu software o una sugerencia que deseen hacer.
- Una herramienta de revisión de código, donde se pueden añadir anotaciones en cualquier punto de un fichero y debatir sobre determinados cambios realizados en un commit específico.
- Un visor de ramas donde se pueden comparar los progresos realizados en las distintas ramas de nuestro repositorio.

¿Qué uso le daremos?

En nuestra especialidad “Programación”, fuimos aprendiendo cosas y creando programas de código abierto, fomentando el software libre; es por eso que presentamos esta gran herramienta enfocada al crecimiento de proyectos comunitarios y libres.

En esta página podremos crear una cuenta gratuita y comenzar a subir repositorios de código (o crearlos desde 0), para que con la ayuda de todos ese proyecto mejore; así como también fortalecer los proyectos de los demás para crecer como grupo.

El repositorio local

Luego de clonar o crear un repositorio el usuario tiene una copia completa del repositorio, y puede realizar operaciones de control de versiones contra este repositorio local, como por ejemplo crear nuevas versiones, revertir cambios, etc. El flujo de trabajo básico en Git es algo así:

- Modificas una serie de archivos en tu directorio de trabajo (working directory).
- Añadís instantáneas de los archivos a tu área de preparación (staging area).
- Confirmas los cambios, lo que toma los archivos tal y como están en el área de preparación, y almacena esa instantánea de manera permanente en tu directorio de Git (git directory).

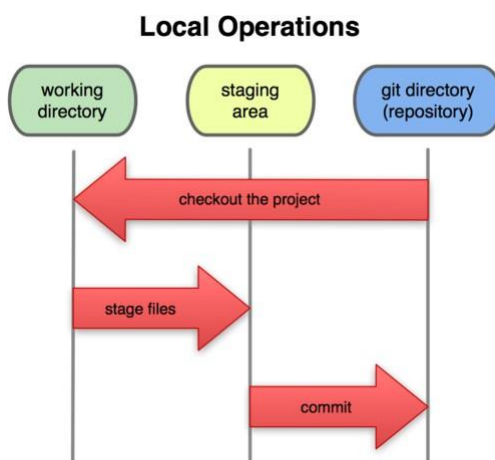


Ilustración 2

Hay dos tipos de repositorios en Git:


- Repositorios “bare”, son generalmente usados en los servidores, aunque puede generarse en cualquier directorio local de quien desee crearlo. Se utiliza para compartir cambios provenientes de diferentes desarrolladores, no se utiliza para crear u compartir cambios desde sí mismo, ya que no cuenta con esas funcionalidades.
- Repositorios de trabajo, permiten crear archivos nuevos, generar cambios a los archivos existentes y a su vez crear nuevas versiones en el repositorio. Si se desea eliminar el repositorio local, sólo debe borrarse la carpeta en la que se encuentra.

Repositorios remotos

Git permite a los usuarios sincronizar el repositorio local con otros repositorios remotos al ambiente de trabajo local. Los usuarios que posean los permisos necesarios pueden hacer un “push” (comando utilizado para subir cambios locales) de los cambios locales al repositorio remoto. A su vez también pueden hacer un “fetch” o “pull” (comandos para buscar cambios) de los cambios realizados en repositorios remotos al local.

Repaso de algunos conceptos útiles

- Repositorio: Un repositorio contiene la historia, las diferentes versiones en el tiempo y todas las diferentes ramas. En Git cada copia del repositorio es un repositorio completo. Si el repositorio en el que estás trabajando no es creado con la opción “bare”, entonces permite hacer un checkout de las revisiones que desees en tu repositorio local.
- Working tree: Posee el contenido de un commit que se puede obtener haciendo un checkout desde un repositorio git. Luego uno puede modificar ese contenido y hacer un nuevo commit con los cambios al repositorio.
- Branch (rama): Un branch es un puntero con un nombre determinado por el usuario que apunta a un commit. Posicionarse en un branch utilizando git es denominado como “hacer un checkout” de ese branch. Si estás trabajando en un determinado branch, la creación de un nuevo commit hace avanzar el puntero a esta nueva instancia. Cada commit conoce sus antecesores así como a sus sucesores en caso de tenerlos. Uno de los branches es el default, generalmente llamado master.



■ **Tag:** Un tag apunta a un commit que unívocamente identifica una versión del repositorio. Con un tag, podés tener un puntero con nombre al que siempre puedas revertir los cambios. Por ejemplo, la versión de 25.01.2009 del branch “testing”.

■ **Commit:** Vos commiteas los cambios a un repositorio. Esto crea un nuevo objeto commit en el repositorio que unívocamente identifica una nueva versión del contenido del repositorio. Esta revisión puede ser consultada posteriormente, por ejemplo si uno quiere ver el código fuente de una versión anterior. Cada commit posee metadata que nos informa acerca del autor, la fecha y otros datos que nos pueden resultar prácticos a la hora de tratar de encontrar uno determinado.

■ **URL:** Una URL en Git determina la ubicación de un repositorio. **Revisión:** Representa una versión del código fuente. Git implementa las revisiones de la misma manera que los objetos commit.

■ **HEAD:** Es un objeto simbólico que apunta generalmente al branch sobre el que estamos trabajando (lo que también conocemos como “checked out branch”). Si uno cambia de un branch al otro el HEAD apunta al último commit del branch seleccionado. Si uno hace un checkout de un determinado commit, el HEAD apunta a ese commit.

■ **Staging area:** Es el lugar en el que se almacenan los cambios del working tree previos al commit. Es decir, contiene el set de cambios relevantes para el próximo commit.

■ **Index:** Es un término alternativo para referirnos al staging area.

Comandos básicos en Git

Obteniendo y creando proyectos

Como ya lo mencionamos previamente, para hacer cualquier cosa con Git, primero hay que tener un repositorio creado, que es donde Git almacena los cambios que haces al código. Hay dos maneras de obtener un repositorio Git. Una forma es simplemente inicializar uno nuevo desde una carpeta existente, como un nuevo proyecto. La segunda forma consiste en clonar un repositorio Git público, como si quisieras una copia o quisieras trabajar con alguien en un proyecto.

Git init - inicializar una carpeta como un repositorio Git

Para crear un repositorio en una carpeta existente de archivos, puedes ejecutar el comando git init en esa carpeta. Por ejemplo, digamos que tenemos una carpeta con algunos archivos adentro, así:

```
$ cd repositorio_nuevo
```

```
$ ls
```

```
README hello.rb
```

Se trata de un proyecto en el que estamos escribiendo ejemplos del “Hola Mundo” en todos los idiomas. Hasta el momento, sólo tenemos el código escrito en Ruby. Para iniciar el control de versión con Git, podemos ejecutar git init.

```
$ git init
```

```
Initialized empty Git repository in /repositorio_nuevo/.git/
```

Ahora se puede ver que hay una subcarpeta oculta llamada “.git” en el proyecto. Este es tu repositorio donde se almacenan todos los cambios del proyecto.

```
$ ls -a
```

```
.      ..      .git  README  hello.rb
```

En pocas palabras, se usa “git init” para convertir una carpeta existente en un nuevo repositorio Git. Se puede hacer esto en cualquier carpeta en cualquier momento.

Nota: Cuando se crea un repositorio con init, la primera vez que haces un push, tienes que correr git push origin master. Además hay que crear el branch master y hacerle un commit con los archivos agregados para que aparezca.

Git clone - copiar un repositorio Git

Si tienes que colaborar con alguien en un proyecto, o si deseas obtener una copia de un proyecto para poder ver o usar el código, debes clonarlo. Para lograrlo sólo tienes que ejecutar el comando git clone [url] con la URL del proyecto que deseas copiar.

```
$ git clone git://github.com/schacon/simplegit.git
```

```
Initialized empty Git repository in /private/tmp/simplegit/.git/
```

```
remote: Counting objects: 100, done.
```

```
remote: Compressing objects: 100% (86/86), done. remote: Total 100 (delta 35), reused 0
(delta 0) Receiving objects: 100% (100/100), 9.51 KiB, done. Resolving deltas: 100%
(35/35), done.
```

```
$ cd simplegit/
```

```
$ ls
```

```
README  Rakefile lib
```


Esto copiará toda la historia de este proyecto, con lo cual lo tendrás a nivel local y te dará una carpeta de trabajo de la rama principal de ese proyecto para que puedas ver el código o empezar a editarlo. Si cambias a la nueva carpeta, puedes ver la subcarpeta .git (ahí es donde están todos los datos del proyecto).

```
$ ls -a
```

```
.      ..      .git  README  Rakefile lib
```

```
$ cd .git
```

```
$ ls
```

```
HEAD  description info      packed-refs
```

```
branches  hooks logs  refs
```

```
config index  objects
```

Por defecto, Git va a crear una carpeta que tiene el mismo nombre que el proyecto en la dirección que le indiques - básicamente cualquier cosa que aparezca después de la última barra de la URL. Si quieres un nombre diferente, puedes ponerlo al final del comando, después de la URL. Entonces, se utiliza el comando `git clone` para obtener una copia local de un repositorio Git, para que puedas verlo y empezar a modificarlo.

Agregando y subiendo cambios

Un concepto que no se debe perder de vista es que Git tiene un índice (index), que funciona como una especie de área de desarrollo (staging area) para los cambios que uno va generando en los archivos a medida que va trabajando. Esto permite terminar de darle forma a los cambios que uno va realizando en su área de trabajo, en lugar de que la herramienta automáticamente te arme los commits.

`git add` - agregar los contenidos de archivos al staging area

En Git tienes que agregar previamente los cambios realizados al staging area para luego poder hacer el commit correspondiente (confirmar los cambios). Si el archivo que estás agregando es nuevo, entonces tienes que correr el comando `git add` para añadirlo inicialmente en tu staging area. Si el archivo ya está en “seguimiento” también tienes que correr el mismo comando (`git add`), no para agregar el archivo, sino para agregar las nuevas modificaciones en tu staging area. Volviendo al ejemplo de Hola Mundo, una vez iniciado el proyecto, empezaremos a agregarle archivos y para ello correremos el comando `git add`. Podemos usar el comando `git status` para ver en qué estado está nuestro proyecto.

```
$ git status -s
```

```
?? README
```

```
?? hello.rb
```

En este momento tenemos dos archivos que **no** están bajo el seguimiento de Git. Ahora podemos agregarlos.

```
$ git add README hello.rb
```

Ahora si corremos el comando git status nuevamente, veremos que fueron agre- gados.

```
$ git status -s
```

```
A    README
```

```
A    hello.rb
```

Es también común agregar en forma recursiva todos los archivos en un nuevo proyecto especificando sólo el directorio de trabajo que se desea agregar, como por ejemplo:

```
$ git add .
```

De esta manera, Git comenzará a agregar todos los archivos que se encuentran en el directorio especificado. En el ejemplo particular que estamos viendo, “git add .” hubiese hecho lo mismo que al escribir:

```
$ git add README hello.rb
```

Para el caso también “git add *” hubiese hecho lo mismo, además de meterse en subdirectorios que estuviesen dentro del directorio actual. Entonces, si ahora editamos uno de estos archivos y corremos el comando “git status” nuevamente, veríamos algo como lo siguiente:

```
$ vim README
```

```
$ git status -s
```

```
AM README
```

```
A    hello.rb
```

El estado ‘AM’ significa que el archivo fue modificado en el disco desde que lo agregamos. Esto significa que si hacemos un commit de nuestro cambios hasta este momento, estaríamos grabando la versión del archivo que teníamos en el momento que corrimos git add, no la versión que está en nuestro disco. Git no asume que lo que uno quiere subir es explícitamente lo que tenemos en el disco, uno le tiene que avisar a Git con el comando git add nuevamente.

En pocas palabras, uno corre git add en un archivo cuando quiere incluir cualquier cambio que le hayas hecho a tu próximo commit. Cualquier cam- bio que no hayas agregado con este comando, no será incluido en el commit, esta particularidad nos ofrece la posibilidad de armar los commits de una forma más certera, con un alto nivel de detalle acerca de qué queremos incluir y qué no.



Referencias Bibliográficas

<https://www.uco.es/aulasoftwarelibre/wp-content/uploads/2015/11/git-cosfera-dia-1.pdf>

<https://git-scm.com/book/es/v2/Git-en-el-Servidor-GitLab>

<https://help.github.com/es>

<https://conociendogithub.readthedocs.io/en/latest/>

Se utilizaron algunas imágenes cuya fuente provienen del sitio [.git-scm.com](https://git-scm.com).

La información troncal del documento se redactó basándose en el sitio oficial de gitref.

