Guia git

2016-02-19

Introducción:

Git es un software de control de versiones diseñado por Linus Torvalds, pensando en la eficiencia y la confiabilidad del mantenimiento de versiones de aplicaciones cuando éstas tienen un gran número de archivos de código fuente.

Existen una serie de buenas practicas que son:

Cada desarrollador o equipo de desarrollo puede hacer uso de Git de la forma que le parezca conveniente. Sin embargo una buena práctica es la siguiente: Se deben utilizar 4 tipos de ramas: Master, Development, Features, y Hotfix.

- **Master**: Es la rama principal. Contiene el repositorio que se encuentra publicado en producción, por lo que debe estar siempre estable.
- **Development**: Es una rama sacada de master. Es la rama de integración, todas las nuevas funcionalidades se deben integrar en esta rama. Luego que se realice la integración y se corrijan los errores (en caso de haber alguno), es decir que la rama se encuentre estable, se puede hacer un merge de development sobre la rama master.
- **Features**: Cada nueva funcionalidad se debe realizar en una rama nueva, específica para esa funcionalidad. Estas se deben sacar de development. Una vez que la funcionalidad esté pronta, se hace un merge de la rama sobre development, donde se integrará con las demás funcionalidades.
- **Hotfix**: Son bugs que surgen en producción, por lo que se deben arreglar y publicar de forma urgente. Es por ello, que son ramas sacadas de master. Una vez corregido el error, se debe hacer un merge de la rama sobre master. Al final, para que no quede desactualizada, se debe realizar el merge de master sobre development.

Las ordenes básicas que tiene son:

- Operaciones básicas:
- Manejo de ramas:
  - Ramas locales
  - Ramas remotas
- Manejo de submodulos:
- Duplicar un repositorio:

Operaciones básicas

Una vez que tienes git instalado y listo para usarse podremos usar los siguientes comandos:

Inicializando un repositorio en un directorio existente

Si tenemos un repositorio con codigo y queremos crear un repositio git tendremos que escribir:

```
$ git init
```

Esto crea un nuevo subdirectorio llamado .git que contiene todos los archivos necesarios del repositorio, un esqueleto de un repositorio Git. Todavía no hay nada en tu proyecto que esté bajo seguimiento.

Añadir archivos

Para empezar a seguir las modificaciones de un fichero habra que usar el comando `git add fichero` y comenzaremos a trackear el archivo “nombre_archivo”.

```
$ git add <nombre_archivo>
```

Si tenemos multiples ficheros y no queremos añadirlos uno a uno podremos hacerlos todos juntos con:

```
$ git add .
```

Confirmar los cambios

Para consolidar el archivo previamente creado y puesto en escenario debemos utilizar el siguiente comando:

```
$ git commit -m 'versión inicial del proyecto'
```

> La bandera -m indica que se debe consolidar el archivo con un mensaje informativo.

Enviar cambios al servidor

Una vez que tenemos los cambios consolidados en nuestro repositorio local, utilizamos el comando `git push` para enviarlo al servidor:

```
$ git push origin master
```

clonacion de un repositorio

Si queremos tener una copia local de un repositorio, el comando que usaremos es `git clone [url]`, tendremos distintas formas de clonar:

- Clonacion de un repositorio

  ```
  git clone git@github.com:procamora/Wiki-Pelican.git
  ```

- Clonacion de un repositorio especificando su nombre

  ```
  git clone git@github.com:procamora/Wiki-Pelican.git wiki
  ```

- Clonacion recursiva con submodulos: `bash git clone --recursive git@github.com:procamora/Wiki-Pelican.git`

  Recuerda que puedes clonar un repositorio por ssh o por https, la diferencia esta en que por ssh no tendras que autenticarte cada vez que realizas un cambio en el servidor.

ver estado del respositorio

Muestra el estado actual de la rama, como los cambios que hay sin commitear.

Si esta todo correcto y no hay ningun cambio la salida sera:

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

Si hay modificaciones pendientes

```
$ git status
On branch master
Untracked files:  #<-- Nos está indicando que tenemos archivos nuevos.
  (use "git add <file>..." to include in what will be committed)
   Archivo2.txt #<-- El archivo nuevo del que git no conoce.
nothing added to commit but untracked files present (use "git add" to track)
```

comparar diferencias

Cuando queremos ver las lineas añadidas y eliminadas de un fichero modificado usaremos:

```
git diff fichero
```

Si quieres ver los cambios que has preparado y que irán en tu próxima confirmación, puedes usar:

```
git diff --cached.
```

Eliminando archivos

Para eliminar un archivo de Git, debes eliminarlo de tus archivos bajo seguimiento (más concretamente, debes eliminarlo de tu área de preparación), y después confirmar. El comando `git rm` se encarga de eso, y también elimina el archivo de tu directorio de trabajo, para que no lo veas entre los archivos sin seguimiento.

Si simplemente eliminas el archivo de tu directorio de trabajo, aparecerá bajo la cabecera "Modificados pero no actualizados" ("Changed but not updated") (es decir, sin preparar) de la salida del comando `git status`:

```
$ rm grit.gemspec
$ git status
# On branch master
#
# Changed but not updated:
#   (use "git add/rm <file>..." to update what will be committed)
#
#       deleted:    grit.gemspec
#
```

Si entonces ejecutas el comando `git rm`, preparas la eliminación del archivo en cuestión:

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

La próxima vez que confirmes, el archivo desaparecerá y dejará de estar bajo seguimiento. Si ya habías modificado el archivo y lo tenías en el área de preparación, deberás forzar su eliminación con la opción `-f`.

Moviendo archivos

A diferencia de muchos otros VCSs, Git no hace un seguimiento explicito del movimiento de archivos. Si renombras un archivo, en Git no se almacena ningún metadato que indique que lo has renombrado. Sin embargo, Git es suficientemente inteligente como para darse cuenta — trataremos el tema de la detección de movimiento de archivos un poco más adelante —.

Por tanto, es un poco desconcertante que Git tenga un comando mv. Si quieres renombrar un archivo en Git, puedes ejecutar algo así:

```
$ git mv README.txt README
```

Sin embargo, esto es equivalente a ejecutar algo así:

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

git log (multiples vistas)

Después de haber hecho varias confirmaciones, o si has clonado un repositorio que ya tenía un histórico de confirmaciones, probablemente quieras mirar atrás para ver qué modificaciones se han llevado a cabo. La herramienta más básica y potente para hacer esto es el comando `git log`.

Por defecto, si no pasas ningún argumento, git log lista las confirmaciones hechas sobre ese repositorio en orden cronológico inverso. Es decir, las confirmaciones más recientes se muestran al principio. Como puedes ver, este comando lista cada confirmación con su suma de comprobación SHA-1, el nombre y dirección de correo del autor, la fecha y el mensaje de confirmación `git log`

Una de las opciones más útiles es -p, que muestra las diferencias introducidas en cada confirmación. También puedes usar la opción -2, que hace que se muestren únicamente las dos últimas entradas del histórico `git log -p -2`.

También puedes usar con git log una serie de opciones de resumen. Por ejemplo, si quieres ver algunas estadísticas de cada confirmación, puedes usar la opción –stat `git log --stat`.

Modificando tu última confirmación

Frecuentemente cuando trabajamos y consolidamos los cambios olvidamos agregar al escenario algún archivo o simplemente lo modificamos tarde y lo queremos agregar a la información consolidada anteriormente. Es aquí cuando entra este comando en acción.

Vamos a realizar una serie de cambios, los consolidaremos y agregaremos otro cambio posteriormente.

```
$ touch README.md
$ echo '# Repositorio git para el curso Git desde cero' >> README.md
$ git status # omitimos la salida
$ git add README.md
$ git commit -m "Agregar archivo README.md"

# Ahora modificaremos otro archivo y lo agregaremos.

$ echo 'Agregamos la tercera linea' >> Archivo2_cambio_de_nombre.txt
$ git add Archivo2_cambio_de_nombre.txt

# En este punto tenemos dos opciones 1) Dejar el mismo mensaje 2) Cambiarlo

$ git commit --amend --no-edit # Dejamos el mismo mensaje
$ git commit --amend -m "Nuevo mensaje para el cambio"
```

> Es sumamente importante que si los cambios fueron consolidados y enviados al servidor remoto NO se utilice este comando; ya que se modifica el número de serial único de los cambios hechos y vamos a tener un conflicto difícil de resolver.

Deshaciendo la preparación de un archivo

Cuando añadimos al escenario un archivo por error, o hacemos un `git add .` cuando no queriamos añadir todos los ficheros con modificaciones tendremos el comodin de eliminar del escenario un fichero o todos con `git reset`

Su funcionamiento es el siguiente: Añadimos todos los ficheros modificados y desacemos el add. `git reset HEAD`

```
$ git add .                 # añadimos todos los ficheros

$ git status                # mostramos el estado actual
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   README.md
        new file:   prueba

$ git reset HEAD            # cancelamos el comando add
Unstaged changes after reset:
M       README.md

$ git status                # mostramos el estado actual
On branch master
Your branch is up-to-date with 'origin/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:   README.md

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

        prueba

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

Su funcionamiento es el siguiente: Añadimos un fichero y desacemos el add de ese unico fichero. `git reset HEAD <archivo>`

```
$ git add prueba                # añadimos el fichero prueba

$ git status                    # mostramos el estado actual
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   prueba

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:   README.md

$ git reset HEAD prueba             # cancelamos el comando add para prueba
Unstaged changes after reset:
M       README.md

$ git status                        # mostramos el estado actual
On branch master
Your branch is up-to-date with 'origin/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:   README.md

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

        prueba

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

Deshaciendo la modificación de un archivo

Si queremos desechar todos los cambios que hemos hecho a un fichero desde el ultimo commit tendremos la opcion `git checkout fichero`

```
$ git status
On branch master
Your branch is up-to-date with 'origin/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:   README.md

no changes added to commit (use "git add" and/or "git commit -a")
$ git checkout README.md

$ git status

On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

Mostrando tus repositorios remotos

Para ver qué repositorios remotos tienes configurados, puedes ejecutar el comando `git remote`. Mostrará una lista con los nombres de los remotos que hayas especificado. Si has clonado tu repositorio, deberías ver por lo menos *"origin"* (es el nombre predeterminado que le da Git al servidor del que clonaste):

```
$ git remote
origin
```

También puedes añadir la opción -v, que muestra la URL asociada a cada repositorio remoto:

```
$ git remote -v
origin  git@github.com:procamora/Wiki-Personal.git (fetch)
origin  git@github.com:procamora/Wiki-Personal.git (push)
```

Añadiendo repositorios remotos

Para añadir un nuevo repositorio Git remoto, asignándole un nombre con el que referenciarlo fácilmente, ejecuta `git remote add [nombre] [url]`:

```
git remote add wiki git://github.com/procamora/Wiki-Personal.git
```

Recibiendo de tus repositorios remotos

Para recuperar datos de los repositorios remotos usamos `git fetch [remote-name]`, aunque normalmente el nombre del repositoio lo podremos saltar ya que solo tendremos uno.

Este comando recupera todos los datos del proyecto remoto que no tengas todavía. Después de hacer esto, deberías tener referencias a todas las ramas del repositorio remoto, que puedes unir o inspeccionar en cualquier momento.

```
$ git fetch
```

Enviando a tus repositorios remotos

Cuando tu proyecto se encuentra en un estado que quieres compartir, tienes que enviarlo a un repositorio remoto. El comando que te permite hacer esto es sencillo: `git push [nombre-remoto][nombre-rama]`. Si quieres enviar tu rama maestra (master) a tu servidor origen (origin), ejecutarías esto para enviar tu trabajo al servidor:

```
$ git push origin master
```

Este comando funciona únicamente si has clonado de un servidor en el que tienes permiso de escritura, y nadie ha enviado información mientras tanto. Si tú y otra persona clonais a la vez, y él envía su información y luego envías tú la tuya, tu envío será rechazado. Tendrás que bajarte primero su trabajo e incorporarlo en el tuyo para que se te permita hacer un envío.

Recibiendo de tus repositorios remotos y actualizando los ficheros

Éste comando es muy similar a `git fetch` aunque realiza dos funciones simultáneas, `git pull` realiza un `git fetch` más un `git merge`. Se descarga los cambios que se encuentren en el repositorio remoto y los unifica con los cambios que tengamos en nuestro equipo.

nota: utilizando este comando puede que ocurran conflictos.

```
$ git pull
```

git tag (multiples formatos)

Git usa dos tipos principales de etiquetas: ligeras y anotadas. Una etiqueta ligera es muy parecida a una rama que no cambia — un puntero a una confirmación específica —. Sin embargo, las etiquetas anotadas son almacenadas como objetos completos en la base de datos de Git. Tienen suma de comprobación; contienen el nombre del etiquetador, correo electrónico y fecha; tienen mensaje de etiquetado; y pueden estar firmadas y verificadas con GNU Privacy Guard (GPG). Generalmente se recomienda crear etiquetas anotadas para disponer de toda esta información; pero si por alguna razón quieres una etiqueta temporal y no quieres almacenar el resto de información, también tiene disponibles las etiquetas ligeras.

Etiquetas anotadas

Crear una etiqueta anotada en Git es simple. La forma más fácil es especificar `-a` al ejecutar el comando `tag`.

El parámetro `-m` especifica el mensaje, el cual se almacena con la etiqueta. Si no se especifica un mensaje para la etiqueta anotada, Git lanza tu editor para poder escribirlo.

```
$ git tag -a v1.4 -m 'my version 1.4'
$ git tag
v0.1
v1.3
v1.4
```

Puedes ver los datos de la etiqueta junto con la confirmación que fue etiquetada usando el comando git show:

```
$ git show v1.4
tag v1.4
Tagger: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Feb 9 14:45:11 2009 -0800

my version 1.4
commit 15027957951b64cf874c3557a0f3547bd83b3ff6
Merge: 4a447f7... a6b4c97...
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sun Feb 8 19:02:46 2009 -0800

    Merge branch 'experiment'
```

Esto muestra la información del autor de la etiqueta, la fecha en la que la confirmación fue etiquetada, y el mensaje de anotación antes de mostrar la información de la confirmación.

Una vez que hayamos creado nuevos tags y queramos compartirlos con el repositorio remoto sólo debemos emplear el comando push con la bandera –tags de la siguiente manera:

```
$ git push origin master --tags
Total 0 (delta 0), reused 0 (delta 0)
To git@github.com:procamora/Wiki-Personal.git
 * [new tag]         v1.4 -> v1.4
```

Etiquetas ligeras

Otra forma de etiquetar confirmaciones es con una etiqueta ligera. Esto es básicamente la suma de comprobación de la confirmación almacenada en un archivo (ninguna otra información es guardada). Para crear una etiqueta ligera no añadas las opciones `-a`, `-s` o `-m`:

```
$ git tag v1.4-lw
$ git tag
v0.1
v1.3
v1.4
v1.4-lw
v1.5
```

Esta vez, si ejecutas el comando git show en la etiqueta, no verás ninguna información extra. El comando simplemente muestra la confirmación.

```
$ git show v1.4-lw
commit 15027957951b64cf874c3557a0f3547bd83b3ff6
Merge: 4a447f7... a6b4c97...
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sun Feb 8 19:02:46 2009 -0800

    Merge branch 'experiment'
```

Ignorando ficheros

Actualiza tu repositorio remoto en caso que algún otro desarrollador haya eliminado alguna rama remota.

```
$ git remote prune origin
```

Elimina los cambios realizados que aún no se hayan hecho commit.

```
$ git reset --hard HEAD
```

Revierte el commit realizado, identificado por el "hash_commit".

```
$ git revert <hash_commit>
```

Manejo de ramas

Una rama Git es simplemente un apuntador móvil apuntando a una de esas confirmaciones. La rama por defecto de Git es la rama master. Con la primera confirmación de cambios que realicemos, se creará esta rama principal master apuntando a dicha confirmación. En cada confirmación de cambios que realicemos, la rama irá avanzando automáticamente. Y la rama master apuntará siempre a la última confirmación realizada. ¿Qué sucede cuando creas una nueva rama? Pues que simplemente se crea un nuevo apuntador para que lo puedas mover libremente.

A la hora de trabajar con ramas tenemos 2 opciones

Como buena práctica dentro de las ramas del árbol es donde deberíamos introducir los cambios a nuestro proyecto y solo luego de comprobar que dichos cambios funcionan y tienen el comportamiento deseado los unimos con el árbol principal. Esto es porque queremos que el árbol se encuentre lo más limpio posible.

Ramas locales

Los comandos que usaremos en este apartado serán:

- git checkout -b
- git branch (múltiples comandos.)
- git merge
- git branch -d

Crear una rama local

El comando `checkout -b nombre_rama` es el comando corto para crear una nueva rama a partir de la rama actual y nos cambia a la misma.

```
$ git checkout -b sbrk
# hemos creado la nueva rama y nos cambiamos a la nueva rama
Switched to a new branch 'sbrk'
```

Comprobar en que rama estamos

El comando `git branch` se usa para el manejo de ramas, tiene múltiples parámetros pero los mas usados por mi son:

Para ver en que rama nos encontramos en este momento habría que usar el comando `git branch`, como resultado obtendremos todas las ramas locales y con un asterisco (*) y en verde la rama en la que nos encontramos actualmente.

```
$ git branch
* master
  sbrk
```

Para ver el ultimo comentario de cada rama del proyecto usaremos el parámetro *-v* `git branch -v`

```
$ git branch -v
* master       44dc25c Create init
  sbrk 44d6g5g add rama
```

Listar todas las ramas locales y remotas.

```
$ git branch -a
* master
  sbrk
  remotes/origin/HEAD -> origin/master
  remotes/origin/master
```

REPASAR git branch -r

Cambiar entre ramas

Para cambiar de una rama a otra usaremos el comando `git checkout nombre_rama`

```
$ git checkout master
D       content/code/2015_12_03_usos_rsync.md
M       theme/plumage
Switched to branch 'master'
```

Unión de ramas

Para unir dos ramas tenemos el comando `git merge nombre_rama`. Hay que tener especial cuidado a la hora de hacer una unión ya que es probable que aparezcan conflictos al intentar unir ambas ramas. Los conflictos ocurren por tratar de unir dos archivos (son el mismo pero están en diferentes ramas) que son diferentes al compararlos linea a linea.

Para realizar una unión el proceso es el siguiente:

1. Nos colocamos en la rama en la que queremos agregar la funcionalidad de la otra rama.
2. Hacemos un merge de la rama que queremos unir `git merge otra_rama`

Por ejemplo si queremos añadir a la rama *master* la funcionalidad desarrollada en la rama *sbrk*

```
$ git checkout master
$ git merge sbrk

# Podemos apreciar que la unión se realizó correctamente y sin conflictos.
Updating 0022f43..bc8fc31
Fast-forward
 README.md | 5 +++++
 1 file changed, 5 insertions(+)
```

Eliminar una rama local

Cuando queremos eliminar una rama usaremos el comando `git branch -d nombre_rama` o `git branch -D nombre_rama` y con esto conseguiremos eliminar la rama.

Es importante diferenciar entre los parámetros `-d` elimina la rama solamente si se ha unido sino nos mandara un error y `-D` elimina la rama aunque no se haya unido.

Una buena practica es eliminar una rama después de unirla a la rama principal

Por ejemplo si queremos eliminar la rama *sbrk* después de haberla unido a la rama principal usaremos el siguiente comando:

```
$ git branch -d sbrk
Deleted branch sbrk (was bc8fc31).
```

Resolver conflictos

Cuando unimos dos ramas (merge) en ocasiones produciremos conflictos entre ficheros por tener el mismo fichero modificado en ramas diferentes. Cuando ocurre eso git nos abrirá con el editor que tengamos configurado por defecto el fichero que tenga el conflicto, tendremos que buscar las lineas que nos introduce git para indicarnos donde esta el problema que son: `<<<<`, `>>>>>` y `======` eliminarlas y después procederemos a arreglar el código o texto adecuadamente, después ya podremos hacer el commit y el push.

```
$ git checkout master
$ git merge sbrk

# Ocurrió un conflicto
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
Automatic merge failed; fix conflicts and then commit the result.
```

Ramas remotas

Una rama remota es una referencia al estado de una rama local en un repositorio remoto.

Las ramas remotas tienen como convención de nombre el siguiente formato: `(remoto)/(rama)`, donde la rama principal en un repositorio remoto se llama `origin/master` y todas las subramas en el repositorio remoto tendrán el prefijo `origin` y del otro lado el nombre de la rama que queramos darle; en nuestro caso podría ser `primera-rama`.

Importante: si no aplicamos el comando `git push origin master` cada vez que queramos respaldar cambios realizados localmente de nuestra rama principal el estado de la rama remota se quedará rezagado.

Crear una rama remota

Para crear una rama remota primero hay que crear una rama local `git checkout -b rama` y despues mandarla al servidor con `git push origin rama`.

Las ramas locales no se sincronizan automáticamente con las remotas, es un comportamiento que se debe realizar manualmente.

Importante: Una rama local puede estar enlazada con una rama remota, pero no tienen porque tener el mismo nombre.

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

# una vez creada la rama vamos a crearla en el repositorio remoto.

$ git push origin sbrk
Total 0 (delta 0), reused 0 (delta 0)
To git@github.com:codeheroco/tutorial-git.git
 * [new branch]      sbrk -> sbrk
```

Eliminar una rama remota

Cuando queremos borrar una rama remota ubicada en el servidor lo haremos con `git push origin :nombre_rama`.

Esto lo que realiza es decirle al servidor que deseche la rama sbrk. El comando es prácticamente idéntico al que utilizamos para crear la rama remota, la única diferencia es que le pasamos el prefijo `:` antes del nombre de la rama.

```
$ git push origin :sbrk
```

Seguimiento de una rama remota

Cuando clonamos un repositorio solo se clona la rama principal `master`, si queremos seguir tambien alguna rama adicional, tendremos que hacerlo manualmente con el comando `git checkout --track origin/<nombre_rama>` o la opcion corta `git checkout -t origin/<nombre_rama>`, si existe una rama remota de nombre `nombre_rama`, al ejecutar este comando se crea una rama local con el nombre `nombre_rama` para hacer un seguimiento de la rama remota con el mismo nombre.

```
$ git checkout --track origin/sbrk
```

Si quisieramos colaborar con una rama remota lo que tendriamos que hacer seria crear una nueva rama local (nueva_rama) y enlazarlo con la rama remota ubicada en el servidor (sbrk):

```
$ git checkout -b nueva_rama origin/sbrk
Branch nueva_rama set up to track remote branch sbrk from origin.
Switched to a new branch 'nueva_rama'
```

Manejo de submodulos

Otras opciones

Cambiar la URI (URL) para un repositorio git remoto

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

Fuente 1

Borrar un commit

```
git reset --hard HEAD~1
```

Ver url repositorios configurados

```
git remote -v
```

Duplicar un repositorio

```
# Make a bare clone of the repository
git clone --bare git@github.com:procamora/old-repository.git

# Mirror-push to the new repository
cd old-repository.git
git push --mirror git@github.com:procamora/new-repository.git

# Remove our temporary local repository
cd ..
rm -rf old-repository.git
```

Fuente 1

Trabajar con submodulos

```
git submodule add git@github.com:procamora/procamora.github.io.git output
```

Fuente 1

Fuentes usadas: wikipedia, librosweb, codehero