Git Tutorial

darthmendez

October 20, 2020

Contents

1 Control de Versiones

El control de versiones es un sistema que registra los cambios realizados en un archivo o un conjunto de archivos a lo largo del tiempo, de modo que se pueden recuperar versiones epecíficas en cualquier momento.

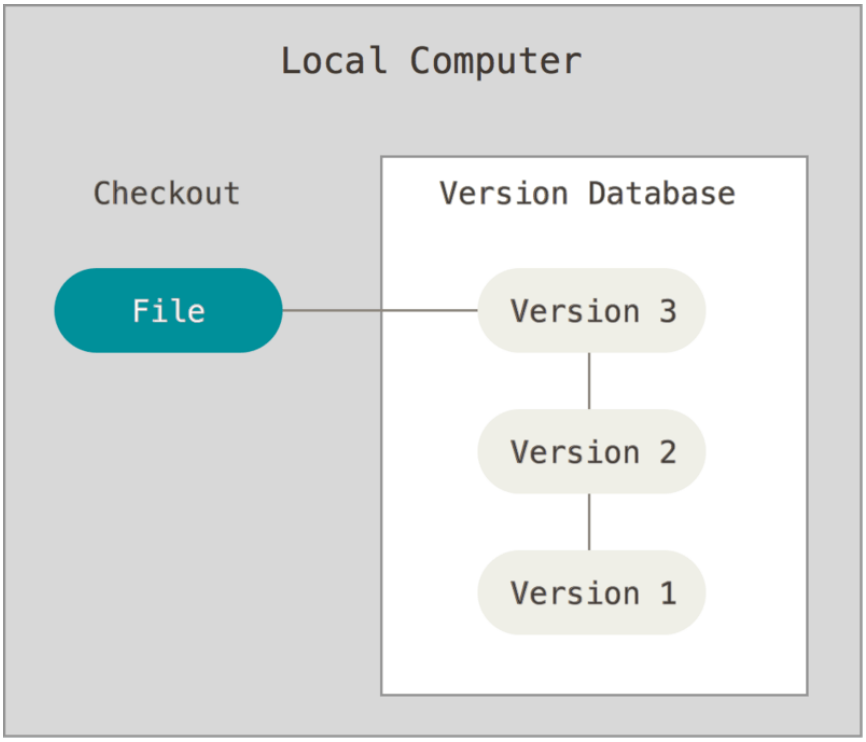
Se puede hacer control de versiones de cualquier cosas que sea un archivo en una computadora.

Los sistemas encargados de realizar dicha gestión se denominan **Sistemas de Control de Verciones** (en inglés, VCS).

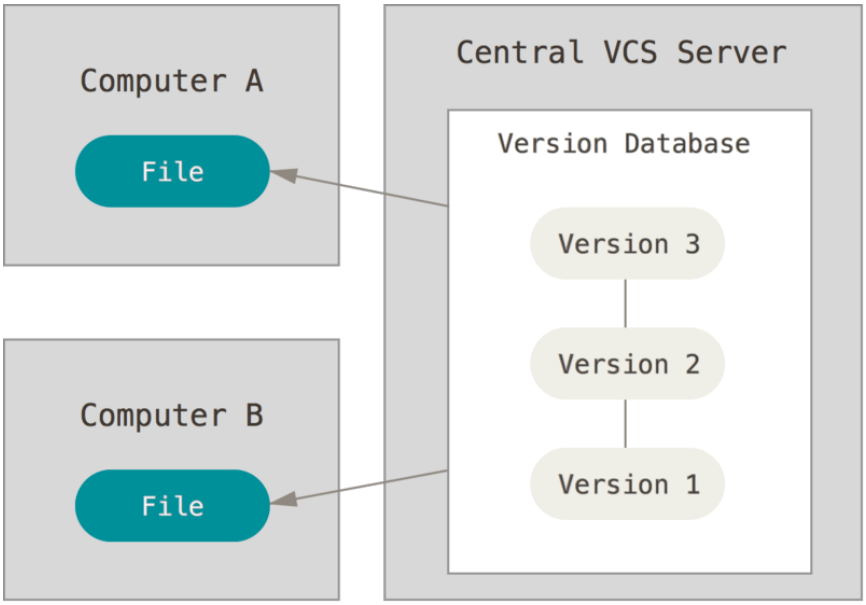
Este apunte se basa en el libro "Progit" de Scott Chacon y Ben Straub, publicado por Apress, con licencia Creative Common Attribution Non Commercial Share. La idea del mismo es ser un apunte de repaso, para mejor manejo del mismo es imprescindible leer el libro.

1.1 Tipos de Control de Versiones

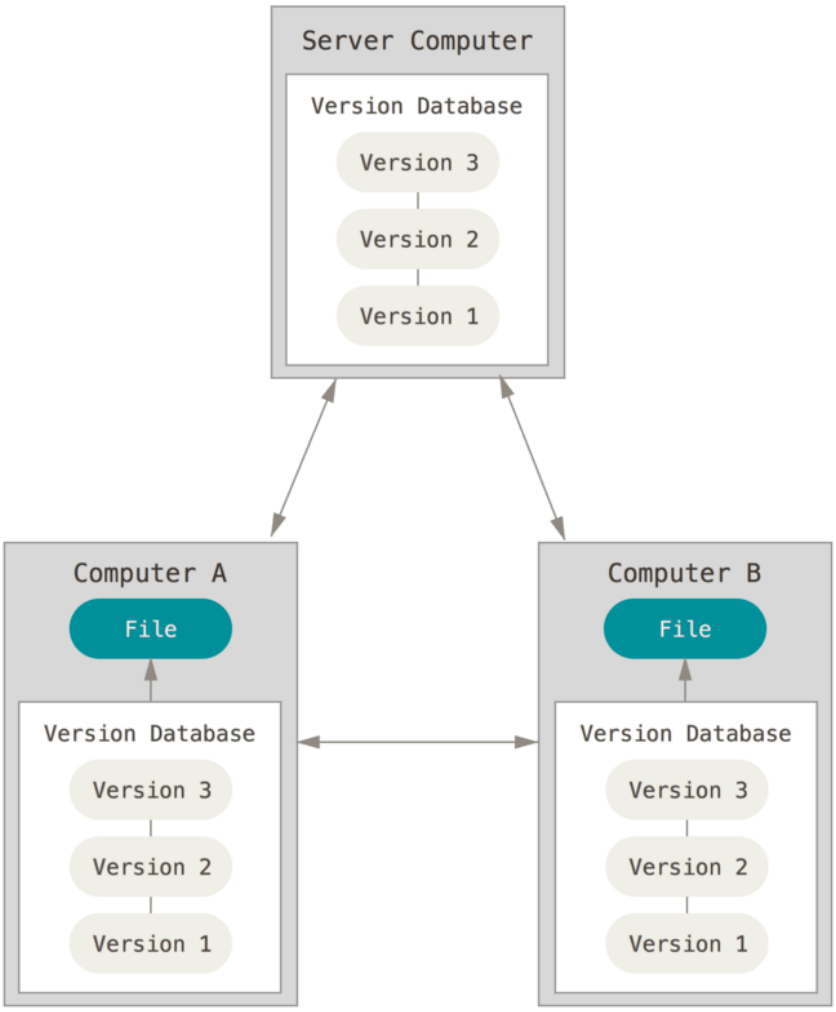
1.1.1 Control de Versiones Locales



1.1.2 Control de Versiones Centralizados



1.1.3 Control de Versiones Distribuidos



1.2 Git

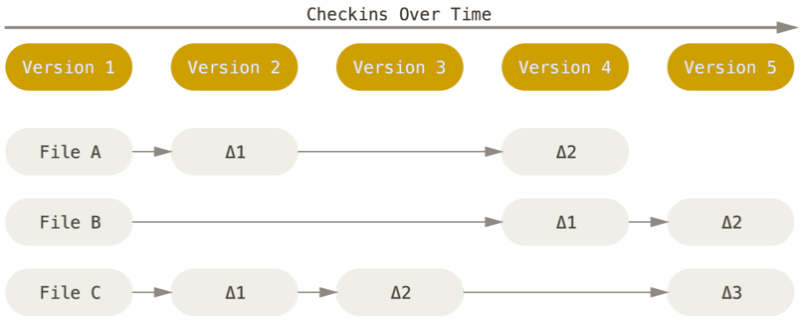
- Git fue creado por Linus Trovals, debido a problemas con el Sistema

de Control de Versiones que se utilizaba para el **Kernel de Linux** llamado BitKeeper.

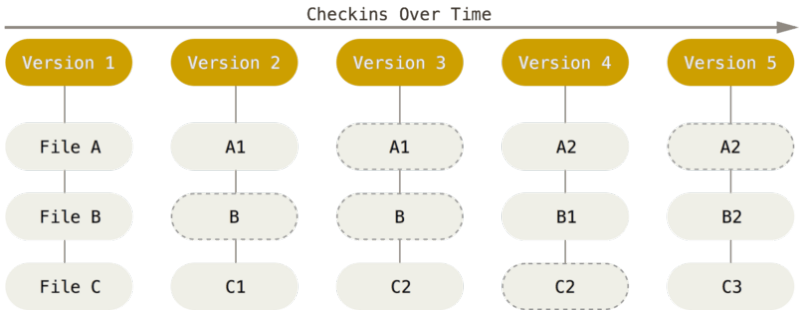
- En 2005 la compañía se fundió y se dejó de ofrecer gratuitamente la herramienta, con lo cual Linus Trovals se decidió a construir su propio Sistema de Control de Versionaes llamado **git**.

1.2.1 Particularidades de Git

- Git hace copias **instantáneas** no diferencias.
- La mayoría de los VCS almacenan la información como una lista de cambios en los archivos:



- Git por el contrario almacena un conjunto de copias instantáneas de un sistema de archivos miniatura. Cada vez que se confirma un cambio se guarda el estado del proyecto en Git. Básicamente toma una foto del aspecto de los archivos en ese momento, si un archivo no se ha modificado no se guarda, sino que guarda un link al que ya se tiene almacenado.



  - Esto hace que Git se parezca más a un sistema de archivos que a un VCS.

1.2.2 Más Particularidades

- Casi todas las operaciones en Git son locales.

- Tiene integridad, todo es verificado mediante una suma de comprobacion antes de ser almacenado, y es identificado a partir de ese momento mediante dicha suma.
- Genera una suma de comprobación que se conose como SHA-1, el hash es una cadena de 40 caracteres

24b9da6552252987aa493b52f8696cd6d3b00373

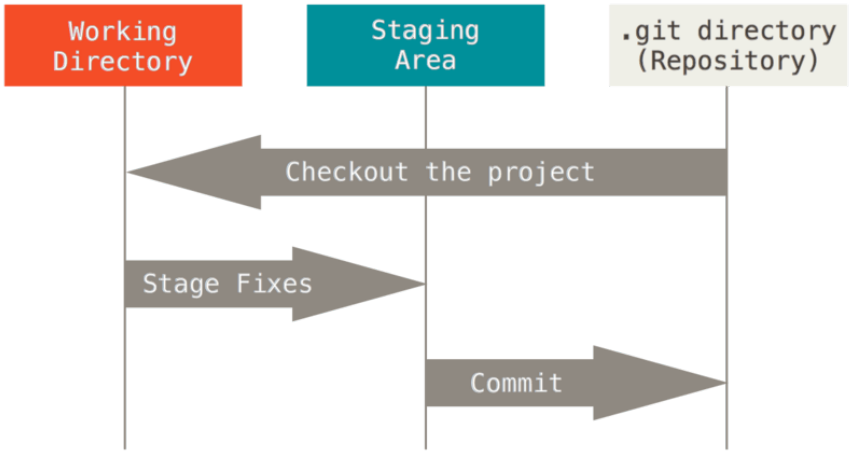
1.3 Los Tres Estados

Los archivos en git pueden tener tres estados:

1. Confirmado (committed): Los datos están almacenados en forma segura en tu base de datos local.
2. Modificado (modified): Un archivo al que se le está haciendo control de versiones ha sido modificado y aun no están confirmados en la base da datos local.
3. Preparado (staged): Se marcó al archivo modificado en su versión actual para que vaya en su próxima confirmación.

Existen por ende tres secciones principales en un proyecto en el que se sigue el control de versiones con Git:

- El directorio de Git (git directory): se almacena la base de datos y los metadatosde objetos de los archivos a los que se hace seguimiento.
- El directorio de Trabajo (working directory): es una copia de una versión del proyecto.
- El área de Preparación (staging area): es un archivo, generalmete contenido en el directorio de Git, que almacena información de lo que va a ir en la próxima Confirmación.



1.3.1 El Flujo de Trabajo Básico

1. Se **modifican** una serie de archivos del directorio de trabajo.
2. Se **preparan** los archivos, añadiéndolos al área de preparación.
3. Se **confirma** los cambios, lo que toma los archivos tal como esán en el área de preparación y hace una copia instantánea que queda en el directorio de Git.

1.4 Configuración Inicial

1.4.1 Una vez instalado Git

hay que configurarle algunos parámetros. Estose puede hacer a mano editando:

- `/etc/gitconfig`
- `~/.gitconfig`
- `config` en el directorio de Git (.git/config)
- Cada nivel sobreescribe al anterior.

1.4.2 Identidad y Editor

1. Identidad

   Setear la identidad del usuario y el email se realiza con el siguiente comando:

```
$ git config --global user.name "Mariano Mendez"
$ git Config --global user.email marianomendez@hotmail.com
```

tener en cuenta que –global indique que será utilizado para todos los proyecyos.

2. Editor

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

3. Comprobar la Configuración

   Para ver la configuracion, se ejecuta el comando `git config --list` :

```
$ git config --list
user.name=John Doe
user.email=johndoe@example.com
color.status=auto
color.branch=auto
color.interactive=auto
color.diff=auto
...
```

1.5 Usar Repositorios

1.5.1 Inicializar un Repositorio

- Para controlor el versionado del contenido de un proyecto o

directorio es necesario entrar en ese directorio y ejecutar:

```
$ git init
```

- Con la ejecución de ese comando git prepara todo para que el

contenido de ese directorio pueda ser versionado. Básicamente creando el directorio `.git` y su estructura.

- Se inicia luego el workflow normal de trabajo:

```
$ git add *.c
$ git add *.h
$ git commit -m 'initial commit'
```

1.5.2 Clonar un Repositorio Existente

Para clonar un repositorio existente es necesario utilizar el comando **git clone [url]**:

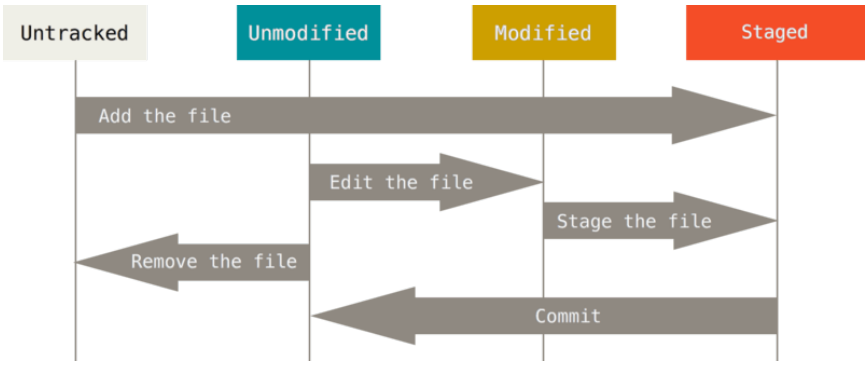
```
$ git clone https://github.com/libgit2/libgit2
```

Con esto se descargará una copia de **todo** el repositorio, con la información del mismmo hasta la fecha.

1.6 Haciendo Cambios en el Repositorio

Una vez que se tiene un repositorio ya creado se está preparado para trabajar en él.

- el ciclo de vida del estado de los archivos de un repositorio se muestra a continuación:



1.6.1 Revisando el Estado de los Archivos

1. status

   Para saber el estado de cada archivo en un determinado repositorio al que se le está haciendo control de cambios, se utiliza el comando **status**:

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

   - Esto significa que el directorio de trabajo está limpio.

- Si se agrega un nuevo archivo al proyecto, que no existía antes de y se verifica **git status** se podra apreciar :

```
$ 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)
```

Se puede apreciar como Git se sabe cuales son los archivos a los cuales no se le esta haciendo *control de versiones*.

1.6.2 Controlar los Cambios de un Nuevo Archivo

Para comenzar a *controlar los cambios de un archivo se debe usar el comando* **git add**.

1. add

   - El comando **add** permite iniciar el control de cambios sobre un

   determinado archivo:

```
$ git add README
```

   - para ver el resultado se ejecuta un **status**

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

    new file:   README
```

1.6.3 Preparar Archivos Modificacados

- Si se realizan cambios en un archivo al cual se le está realizando control de cambios o versiones, por ejemplo un archivo llamado CONTRIBUTING.md y se chequea el status del proyecto se verá:

```
$ 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
```

- Es decir que Git se dió cuenta que el archivo ha sido modificado y

además que el mismo no ha sido preparado para la próxima confirmación.

- Para hacer que el mismo sea tenido en cuenta en la

próxima confirmación se debe pasar el mismo a la **staging area** o al area de preparación, esto se hace ejecutando un **git add**:

```
$ 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
```

- de esta forma ya esta "**prerarado**" para ser incluído en la próxima confirmación (commit).

1.6.4 Modificar un Archivo Preparado

puede suceder que se realicen modificaciones a un archivo que ya estaba preparado en la **staging area**, con lo cual se veria algo por el estilo al revisar el stado del proyecto:

```
$ 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
```

- Se vé que se tiene el mismo archivo en el **staging area** y el **working directoru** marcado con cambios y no preparado.
- La solución es volverlo a preparar para que entre en el próximo commit:

```
$ 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
```

- Con esto ya se tienen todos los cambios preparados (**staged**) para la próxima confirmacion (**commit**).

1.6.5 Ignorar Archivos

- Es prosible que no se quiera realizar el control de cambios de algunos archivos. Por ejemplo, archivos intermedios de compilación.

- Para ello existe un archivo en el que se puede especificar, mediante expresiones regulares, que tipo de archivos no deben ser considerados por Git. Este archivo se llama **.gitignore**:

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

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

1.6.6 Ver Cambios

- Para ver los cambios que aun no se preparanron se debe ejecutar **git diff** sin parámetros. Ojo! solo se ven los cambios no preparados.
- Para ver los cambios preparados y que se van a incluir en el próximo commit se utiliza el comando **git diff –staged**
- para ver los cambios entre dos commits debe ejecutarse **git diff** $\mathbf{hash_{commit1}}$ $\mathbf{hash_{commit2}}$.

1.6.7 Confirmar los Cambios

Para confirmar cambios que estan en el área de preparación (staging area) se utiliza el comando:

```
$ git commit
```

- En este caso abrirá el editor de texto predeterminado y permitirá escribir un mensaje para el commit.

- Puede abreviarse utilizando la opcion -m y entre comillas dobles el mensaje, no es muy recomendable.

1.6.8 Comandos sobre Archivos

- una vez que se está realizando control de versiones existen algunos comando que nos permiten manejar archivos:

1. git rm

   - Este comando elimina un archivo al cual se le está haciendo control de versiones. Lo elimina del disco!

     ```
     $ git rm CONTRIBUTING.md
     ```

   - Si se quiere eliminar el archivo del control de versiones pero no del disco se debe ejecutar el comando:

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

2. git mv

   Este comando permite cambiar el nombre de un archivo trackeado.

1.7 Historial de Versinones

Para ver el historial de confirmaciones o commits se debe utilizar el comando git log:

```
$ git log
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

 changed the version number

 commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
 Author: Scott Chacon <schacon@gee-mail.com>
 Date:   Sat Mar 15 16:40:33 2008 -0700

 removed unnecessary test

 commit a11bef06a3f659402fe7563abf99ad00de2209e6
```

```
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700

first commit
```

Este comando tiene muchas opciones :

- `-p`: muestra las difefrencias introducidas en cada commit
- `--stat`: muestra estadisticas de cada commit
- `--shortstat`: muestra solo la lines de resumen de la opción `--stat`
- `--pretty`: modifica el formato de salida:
    1. oneline
    2. short
    3. full
    4. fuller
- `--graph`: Añade un pequeño gráfico ASCII

```
$ git log --pretty=format:"%h %s" --graph
* 2d3acf9 ignore errors from SIGCHLD on trap
*  5e3ee11 Merge branch 'master' of git://github.com/dustin/grit
,|\
,| * 420eac9 Added a method for getting the current branch.
* | 30e367c timeout code and tests
* | 5a09431 add timeout protection to grit
* | e1193f8 support for heads with slashes in them
,|/
* d6016bc require time for xmlschema
*  11d191e Merge branch 'defunkt' into local
```

1.8 Deshacer Cosas

- Cuando se desee deshacer cosas se debe utilizar el comando –**amend**. Hay que tener cuidado ya que a veces no es posible recuperar despues de haber deshecho.

- Por ejemplo si se confirman una serie de casos y se de desea incluir un archivo al que se le realizaron cambios, podría hacerse lo siguiente:

```
$ git commit -m 'initial commit'
$ git add forgotten_file
$ git commit --amend
```

Con lo cual se terminará con una sola confirmación, la segunda confirmacion se reemplaza por la primera.

1.8.1 Deshacer un Archivo Preparado

- Dado que Git siempre dá una ayuda de lo que se puede hacer, en el siguiente ejemplo se puede ver:

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

    renamed:    README.md -> README
    modified:   CONTRIBUTING.md
```

- En este caso se agregó al área de preparación todo un directorio y el comando git status muestra como están las tres áreas y ademas da una idea de una acción posible a realizar que es eliminar de la staging area un archivo, con el comando **git reset HEAD <file> ....**.

- Lo que hace este comando es es **deshacer la preparación de un archivo** :

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

    renamed:    README.md -> 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 vuelve a estar **modificado** pero no preparado.

1.8.2 Deshacer un Archivo Modificado

- Si no se quieren mantener los cambios de realizados en un

determinado archivo (por cuestión de practicidad CONTRIBUTING.MD), se puede volver atrás al estado que estaba en **la última conformación o commit** o al estado de recien clonado:

```
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
```

- Puede verse que se indica en el mismo mensaje como descartarlos cambios realizado en el working directory.

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

    renamed:    README.md -> README
```

- Cabe tener en cuenta que la ejecución de este comando hará que cualquier cabio realizado al archivo desaparecerá, y se perderá.

- Existe una forma para mantener y deshacer temporalmente los cambios, se denomina **stashing**.

1.9 Trabajar con repositorios remotos

Existen varios sitios web que hostean repositorios git, por ejemplo, gitHub o gitLab. Estos proveen cuentas en las que un usuario se registra y puede disponer de espacio para repositorios git. Esto es una de las ventajas de trabajar con git ya que se ha convertido en el estándar **de-facto** para realizar gestion de versiones en la actualidad.

1. Repositorio remoto

    - Normalmente uno se crea una cuenta en alguno de estos sitios y esto le permite crear repositorios a los que podra acceder mediante git.

1.9.1 Clonar un repositorio remoto

- Normalmente cuando se crea un repositorio remoto para traer los datos del mismo se debe **clonarlo**. Para ello se utiliza el comando **git clone <url>**

```
$ git clone https://github.com/mendezmariano/st.git
Cloning into 'st'...
remote: Enumerating objects: 47, done.
remote: Counting objects: 100% (47/47), done.
remote: Compressing objects: 100% (27/27), done.
remote: Total 47 (delta 26), reused 41 (delta 20), pack-reused 0
Unpacking objects: 100% (47/47), 62.30 KiB | 341.00 KiB/s, done.
```

    - Este comando trae el repositorio remoto, con tooodo su historial, a la máquina local en la que se está trabajando.
    - Git siempre le llama al repositorio remoto **origin**

1.10 Manejando Repositorios Remotos

1. Listar Repositorios Remotos

    - Para listar los repositorios remotos se utiliza el comando **git remote**:

```
$ git remote -v
origin https://github.com/schacon/ticgit (fetch)
```

```
origin https://github.com/schacon/ticgit (push)
pb     https://github.com/paulboone/ticgit (fetch)
pb    https://github.com/paulboone/ticgit (push)
```

2. Enviar Cambios Locales al Repositorio Remoto

   - Una vez que se está trabajando con un repositorio remoto, normalmente se desean enviar cambios locales al mismo. Esto se realiza con el comando **git push**, normalmente despues de una confirmación o commit:

```
$ git push origin main
```

3. Recibir Cambios Locales al Repositorio Remoto

   - Para recibir los cambios que se han pusheado (enviados) por otros al repositorio remotos, se utiliza el comando **git pull**. Es importante, tener el ábito cuando se trabaja con repositorios remotos es hacer siempre un pull.

4. Renombrar un Remoto

   - Para renombrar un repositorio remoto se utiliza el comando **git remote rename** $<\textbf{nombre}_{\textbf{viejo}}>$ $<\textbf{nombre}_{\textbf{nuevo}}>$

```
$ git remote rename pb paul
$ git remote
origin
paul
```

5. Eliminar un Remoto

   - Para ello se utiliza el comando **git remote rm** $<$**nombre**$>$:

1.11 TODO Etiquetas

1.11.1 TODO Etiquetas

- Ligeras
- Anotadas

1.11.2 TODO Crear Etiquetas

1. **TODO** Ligeras
2. **TODO** Anotadas
3. **TODO** Etiquetado Tardio

1.12 Ramificaciones o Branches