

Git: Desarrollo colaborativo

Módulo 5

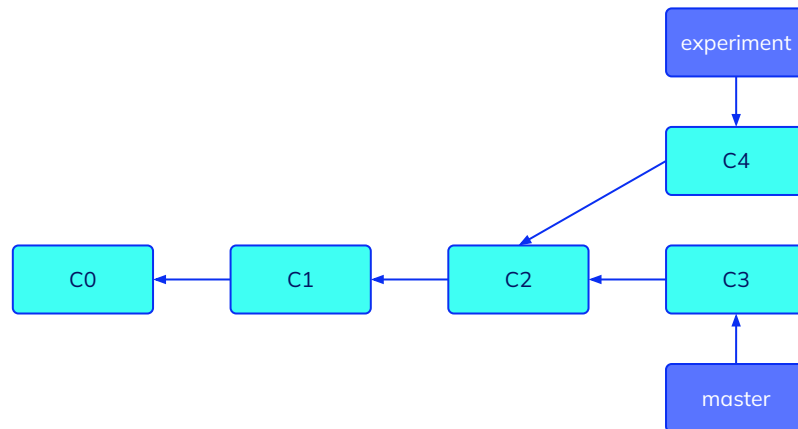
Introducción a Rebase

Git rebase

¿Qué pasaría si pudiéramos **ordenar nuestros *branches* bifurcados** de forma tal que en vez de separar su historial desde un padre en común, pudieran formar una **estructura de grafo lineal**, es decir, como si nunca se hubieran bifurcado? Intentemos imaginar ese escenario...



Historial



En el ejemplo del slide anterior, se podrían reubicar los cambios introducidos en el `commit` C4 y aplicarlos encima del `commit` C3. Como comentamos previamente, si este fuera el caso, en lugar de tener C3 y C4 en caminos distintos, tendríamos a C4 por delante de C3 dándonos el historial C0 - C1 - C2 - C3 - C4.

De esta manera, se puede volver a tener **una estructura lineal de trabajo**, lo que permite **interpretar mejor y más rápido nuestro historial**.

Para realizar esta operación se puede utilizar el comando `git rebase`, que nos permite llevar los cambios confirmados en un *branch* y aplicarlos sobre otro en vez de fusionar a tres bandas.

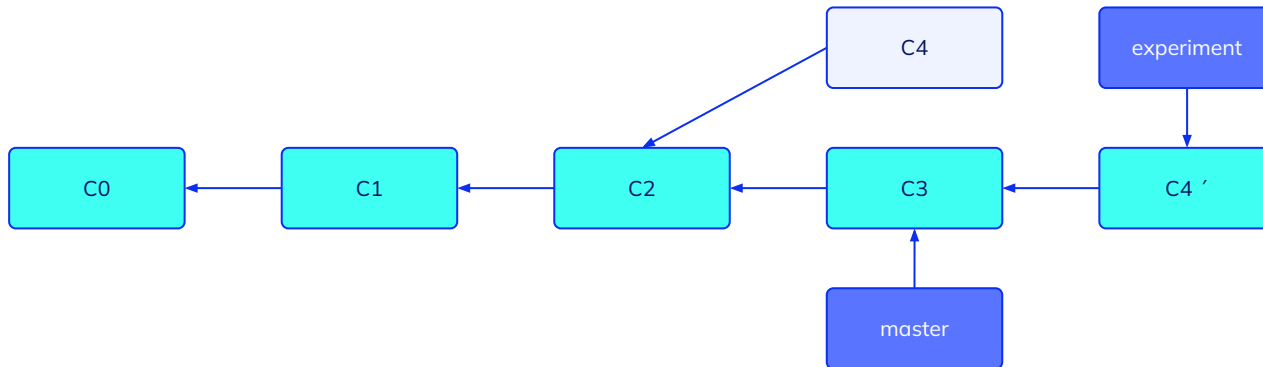
Para esto, nos dirigimos a aquel *branch* que se había bifurcado y que queremos tener de manera lineal y luego corremos el siguiente comando:

```
> git rebase <branch>
```



De esta manera, indicamos después de qué rama queremos ver sus cambios:

```
> git checkout experiment  
> git rebase master  
First, rewinding head to replay your work on top of it...  
Applying: added staged command
```



Pasos

Hacer que Git vaya al ancestro común de ambas ramas (donde estás actualmente y de donde quieres reorganizar):

1. Sacar las diferencias introducidas por cada confirmación en la rama donde estás.
2. Guardar esas diferencias en archivos temporales.
3. Reiniciar (reset) la rama actual hasta llevarla a la misma confirmación en la rama de donde quieres reorganizar.
4. Volver a aplicar ordenadamente los cambios.

En este momento, puedes volver a la rama master y hacer una fusión con avance rápido (**fast-forward merge**).



Merge vs. Rebase: contras y beneficios

No alcanzamos la dicha del rebase sin sus contrapartidas, que pueden resumirse en una línea:

Nunca reorganices confirmaciones de cambio (**commits**) que hayas enviado (**push**) a un repositorio público.


Si sigues esta recomendación, no tendrás problemas. Pero si no lo haces podrás causar problemas en el repositorio, generando conflictos en el resto de tu grupo de trabajo.

Cuando se le hace **rebase** a algo, se están abandonando las confirmaciones de cambio ya creadas y se están creando nuevas, que son similares, pero diferentes.

Si envías (**push**) confirmaciones (**commits**) a alguna parte, y otros las recogen (**pull**) de allí; y luego las reescribes con **git rebase** y las vuelves a enviar (**push**); tus colaboradores tendrán que re fusionar (**re-merge**) su trabajo y todo se volverá tremendamente complicado cuando intentes recoger (**pull**) su trabajo de vuelta sobre el tuyo.


Para algunos desarrolladores, **el historial de confirmaciones de tu repositorio es un registro de todo lo que ha pasado**. Un documento histórico, valioso por sí mismo y que no debería ser alterado. Desde este punto de vista, cambiar el historial de confirmaciones es casi como blasfemar.

Estarías mintiendo sobre lo que en verdad ocurrió. ¿Y qué pasa si hay una serie desastrosa de fusiones confirmadas? Nada. Así fue como ocurrió y el repositorio debería tener un registro de esto para la posteridad.



La otra forma de verlo es que el historial de confirmaciones es la historia de cómo se hizo tu proyecto. Tú no publicarías el primer borrador de tu novela, y el manual de cómo mantener tus programas también debe estar editado con mucho cuidado.

Esta es el área que utiliza herramientas como rebase y filter-branch para contar la historia de la mejor manera para los futuros lectores.



La respuesta a qué es mejor si merge o rebase no es tan sencilla. Git es una herramienta poderosa que permite hacer muchas cosas con el historial, y cada equipo y proyecto es diferente.

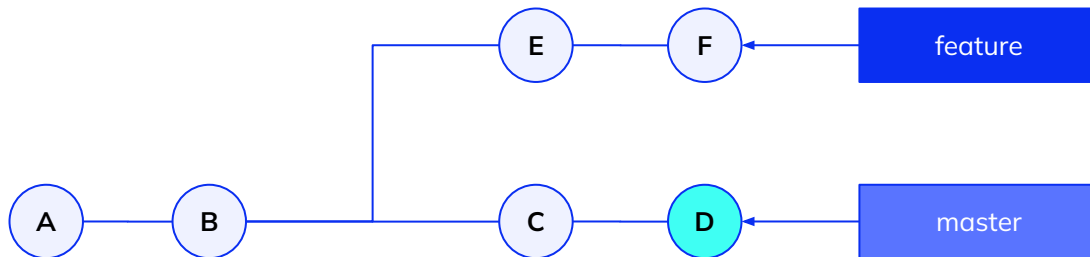
Ahora que conoces cómo trabajan ambas herramientas, podrás decidir cuál de las dos es mejor para tu caso particular.

Normalmente, **la manera de sacar lo mejor de ambas es reorganizar el trabajo local, que aún no has compartido, antes de enviarlo a algún lugar; pero nunca reorganizar nada que ya haya sido compartido.**



El comando Cherry-pick

Imaginemos que estamos trabajando sobre dos *branches* distintos: **master** y **feature**, y que queremos usar el **commit** C del *branch master* dentro de nuestro *branch feature*:

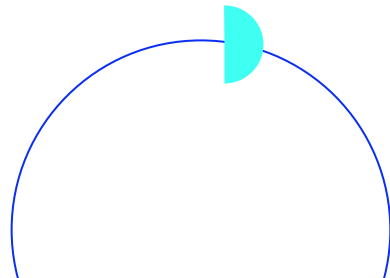


Pasos a seguir para realizar este trabajo:

1. Obtener el **hash** del **commit**. Podemos hacer esto de, al menos, dos maneras:
 - Utilizar el comando **git log --online**, para obtener todo el historial de **commits**. Asegurarnos de estar en el *branch* correcto, en este caso: **master**, ya que queremos usar un **commit** de ahí.
 - Ir a GitHub y seleccionar el **hash** del **commit** desde ahí.
 - Hacer un **git checkout** al *branch* en el que se aplicará dicho **commit**; en este caso: **git checkout feature**.

2. Utilizar el comando **git cherry-pick**:

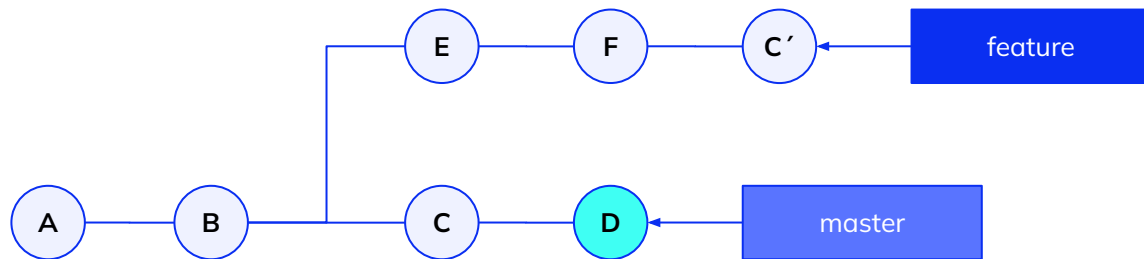
```
> git cherry-pick C
```



Si corremos luego el comando **git log** se puede observar que aplicamos un **commit** nuevo, pero con los cambios viejos del *branch master* a lo último de nuestro *branch feature*:



Se debe tener en cuenta que el **commit** nuevo agregado por encima del *branch feature* va a tener un **hash** distinto al **hash** que originalmente tenía en el *branch master*, pero los cambios van a ser los mismos en ambos lados del proyecto.



Stash


El **stash** toma nuestro estado “sucio” de trabajo del **Working Directory**, es decir, las modificaciones de archivos en seguimiento y cambios ya listos que forman parte del **Stage Area**, y los guarda en una pila de cambios sin terminar, que podemos aplicar luego, en cualquier momento de nuestro desarrollo, sobre el mismo *branch* o uno distinto.

Crear un primer stash

Para demostrar qué es el **stash**, imaginemos que estamos desarrollando un proyecto y comenzamos a trabajar en varios archivos, posiblemente preparando cambios.



Si ejecutamos el comando **git status** podríamos ver un estado “sucio” de trabajo:





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

    modified:   index.html

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:   lib/simplegit.rb
```



Ahora necesitamos **cambiar de *branch***, pero no queremos generar un **commit** con lo que venimos trabajando hasta el momento, entonces se pueden guardar los cambios de manera temporal con el comando **git stash** o **git stash push**:



```
> git stash
Saved working directory and index state \
  "WIP on master: 049d078 added the index file"
HEAD is now at 049d078 added the index file
(To restore them type "git stash apply")
```

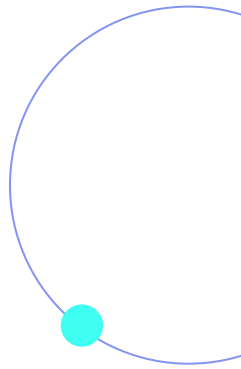


Se puede observar que el **Working Directory** está “limpio”:

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

En este punto, podemos tranquilamente cambiar de *branch* dado que las modificaciones temporales quedaron almacenadas en una pila. Incluso, es posible ver qué cambios hemos guardado usando el comando **git stash list**:

```
> git stash list
stash@{0}: WIP on master: 049d078 added the index file
stash@{1}: WIP on master: c264051 Revert "added file_size"
stash@{2}: WIP on master: 21d80a5 added number to log
```



Restaurar trabajo desde el Stash

Podemos volver a aplicar cualquier modificación temporal que hubiéramos guardado en el **stash** usando el comando **git stash apply**:

```
> git stash apply
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#
#       modified:   index.html
#       modified:   lib/simplegit.rb
#
```



Si se ejecuta **git stash apply**, tal cual el ejemplo anterior, vamos a notar que se restauran únicamente los últimos cambios que se guardaron en el **stash** sin la posibilidad de elegir uno de los que habíamos guardado con anterioridad.

Si, en cambio, se desean aplicar los cambios de un **stash** más viejo, se especifica con su nombre de referencia:

```
> git stash apply stash@{2}
```



**¡Sigamos
trabajando!**