

# Ramificaciones en Git

Cualquier sistema de control de versiones moderno tiene algún mecanismo para soportar el uso de ramas. Cuando hablamos de ramificaciones, significa que tú has tomado la rama principal de desarrollo (master) y a partir de ahí has continuado trabajando sin seguir la rama principal de desarrollo. En muchos sistemas de control de versiones este **proceso es costoso, pues a menudo requiere crear una nueva copia del código**, lo cual puede tomar mucho tiempo cuando se trata de proyectos grandes.

Algunas personas resaltan que **uno de los puntos más fuertes de Git** es su sistema de ramificaciones y lo cierto es que esto le hace resaltar sobre los otros sistemas de control de versiones. ¿Por qué esto es tan importante? La forma en la que Git maneja las ramificaciones es increíblemente rápida, haciendo así de las operaciones de ramificación algo casi instantáneo, al igual que el avance o el retroceso entre distintas ramas, lo cual también es tremadamente rápido. A diferencia de otros sistemas de control de versiones, Git promueve un ciclo de desarrollo donde las ramas se crean y se unen ramas entre sí, incluso varias veces en el mismo día. Entender y manejar esta opción te proporciona una poderosa y exclusiva herramienta que puede, literalmente, cambiar la forma en la que desarrollas.

## ¿Qué es una rama?

Para entender realmente cómo ramifica Git, previamente hemos de examinar la forma en que almacena sus datos.

Recordando lo citado en [Inicio - Sobre el Control de Versiones](#), Git **no los almacena de forma incremental (guardando solo diferencias)**, sino que los almacena como una serie de instantáneas (copias puntuales de los archivos completos, tal y como se encuentran en ese momento).

En cada confirmación de cambios (commit), Git almacena una instantánea de tu trabajo preparado. Dicha instantánea contiene además unos metadatos con el autor y el mensaje explicativo, y uno o varios apuntadores a las confirmaciones (commit) que sean padres directos de esta (un parent en los casos de confirmación normal, y múltiples padres en los casos de estar confirmando una fusión (merge) de dos o más ramas).

Para ilustrar esto, vamos a suponer, por ejemplo, que tienes una carpeta con tres archivos, que preparas (stage) todos ellos y los confirmas (commit). Al preparar los archivos, Git realiza una suma de control de cada uno de ellos (un resumen SHA-1, tal y como se mencionaba en [Inicio - Sobre el Control de Versiones](#)), almacena una copia de cada uno en el repositorio (estas copias se denominan "blobs"), y guarda cada suma de control en el área de preparación (staging area):

```
$ git add README test.rb LICENSE  
$ git commit -m 'initial commit of my project'
```

Cuando creas una confirmación con el comando **git commit**, Git realiza sumas de control de cada subdirectorios (en el ejemplo, solamente tenemos el directorio principal del proyecto), y las guarda como objetos árbol en el repositorio Git. Después, Git crea un objeto de confirmación con los metadatos pertinentes y un apuntador al objeto árbol raíz del proyecto.

En este momento, el repositorio de Git contendrá cinco objetos: un "blob" para cada uno de los tres archivos, un árbol con la lista de contenidos del directorio (más sus respectivas relaciones con los "blobs"), y una confirmación de cambios (commit) apuntando a la raíz de ese árbol y conteniendo el resto de metadatos pertinentes.

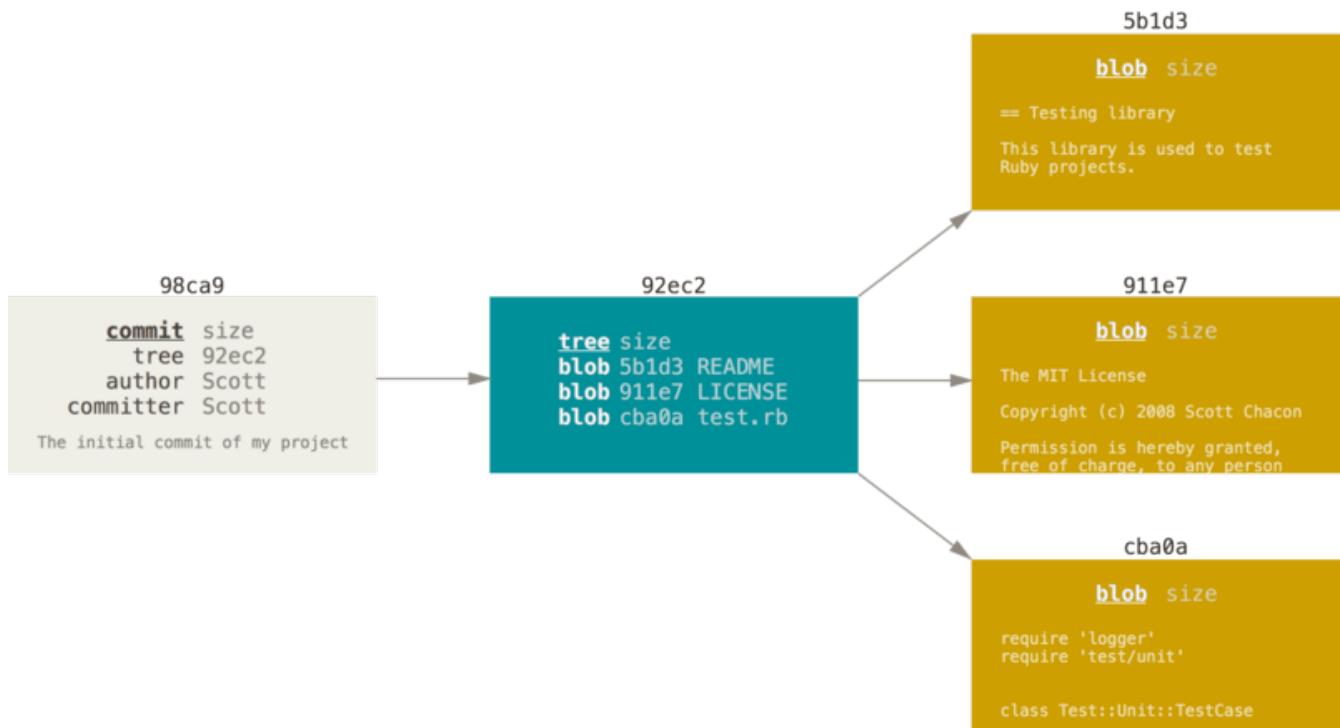


Figure 9. Una confirmación y sus árboles

Si haces más cambios y vuelves a confirmar, la siguiente confirmación guardará un apuntador su confirmación precedente.

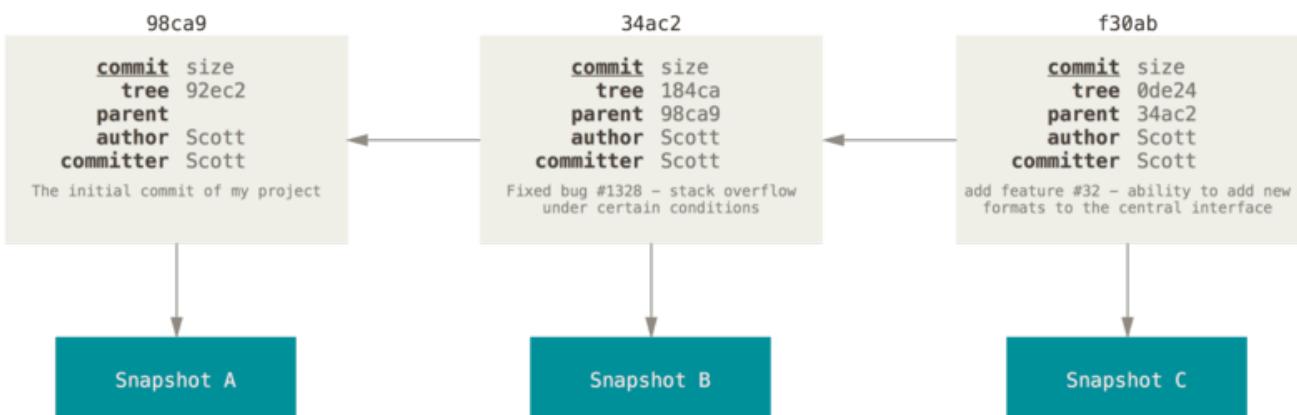


Figure 10. Confirmaciones y sus predecesoras

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.

**NOTE**

La rama “master” en Git no es una rama especial. Es como cualquier otra rama. La única razón por la cual aparece en casi todos los repositorios es porque es la que crea por defecto el comando `git init` y la gente no se molesta en cambiarle el nombre.

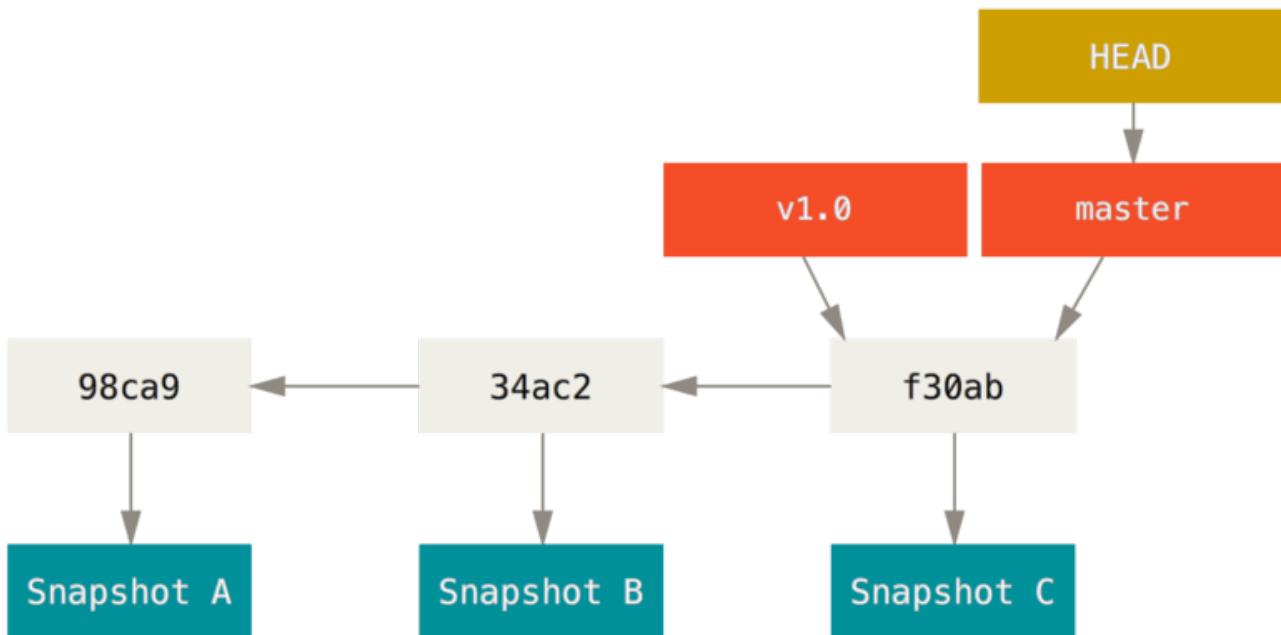


Figure 11. Una rama y su historial de confirmaciones

## Crear una Rama Nueva

¿Qué sucede cuando creas una nueva rama? Bueno..., simplemente se crea un nuevo apuntador para que lo puedas mover libremente. Por ejemplo, supongamos que quieres crear una rama nueva denominada "testing". Para ello, usarás el comando `git branch`:

```
$ git branch testing
```

Esto creará un nuevo apuntador apuntando a la misma confirmación donde estés actualmente.

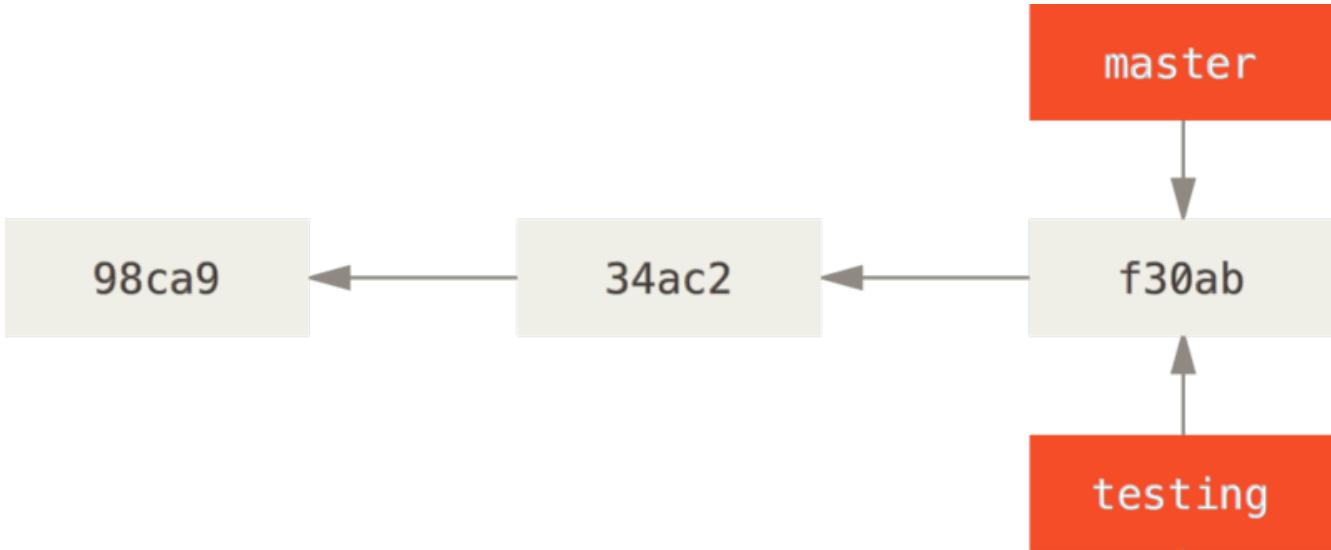


Figure 12. Dos ramas apuntando al mismo grupo de confirmaciones

Y, ¿cómo sabe Git en qué rama estás en este momento? Pues..., mediante un apuntador especial denominado **HEAD**. Aunque es preciso comentar que este HEAD es totalmente distinto al concepto de HEAD en otros sistemas de control de cambios como Subversion o CVS. En Git, es simplemente el apuntador a la rama local en la que tú estés en ese momento, en este caso la rama **master**; pues el comando `git branch` solamente crea una nueva rama, y no salta a dicha rama.

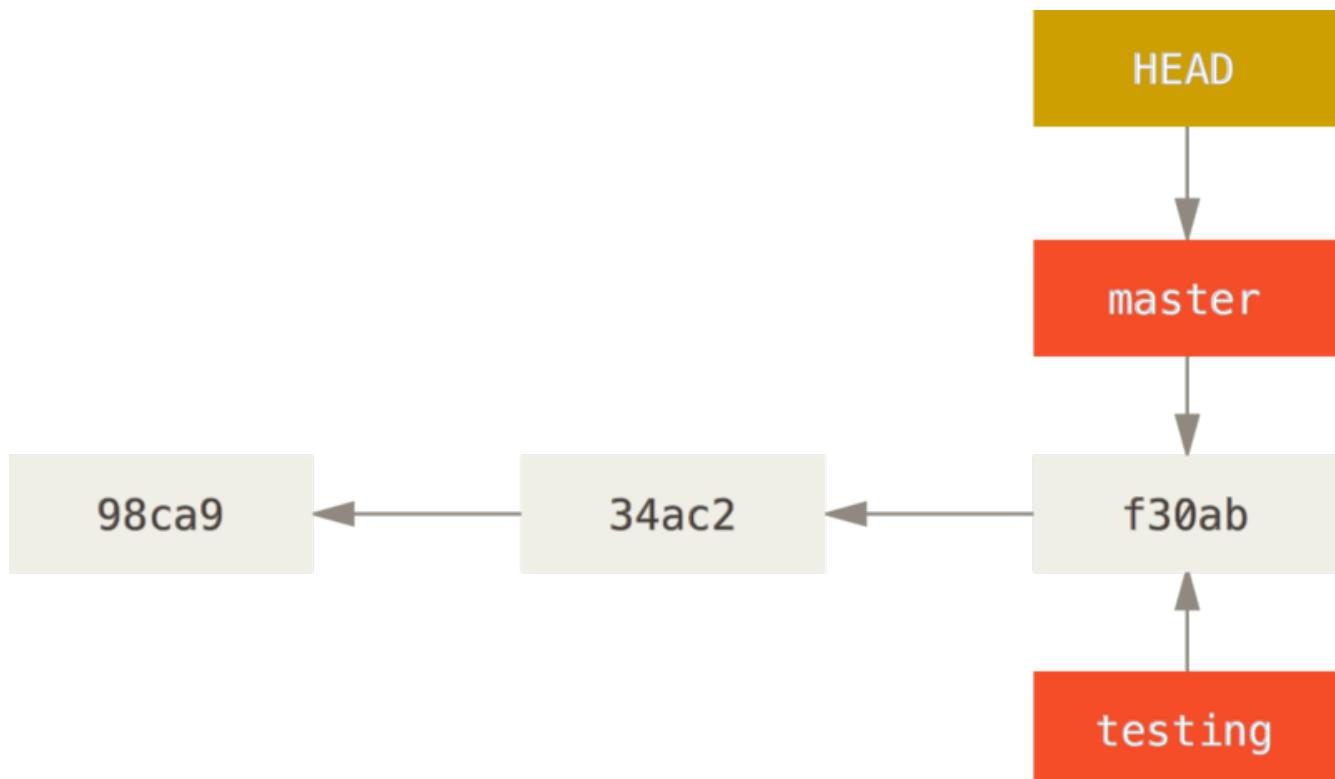


Figure 13. Apuntador HEAD a la rama donde estás actualmente

Esto puedes verlo fácilmente al ejecutar el comando `git log` para que te muestre a dónde apunta cada rama. Esta opción se llama **--decorate**.

```
$ git log --oneline --decorate
f30ab (HEAD, master, testing) add feature #32 - ability to add new
34ac2 fixed bug #1328 - stack overflow under certain conditions
98ca9 initial commit of my project
```

Puedes ver que las ramas “master” y “testing” están junto a la confirmación `f30ab`.

## Cambiar de Rama

Para saltar de una rama a otra, tienes que utilizar el comando `git checkout`. Hagamos una prueba, saltando a la rama `testing` recién creada:

```
$ git checkout testing
```

Esto mueve el apuntador HEAD a la rama `testing`.

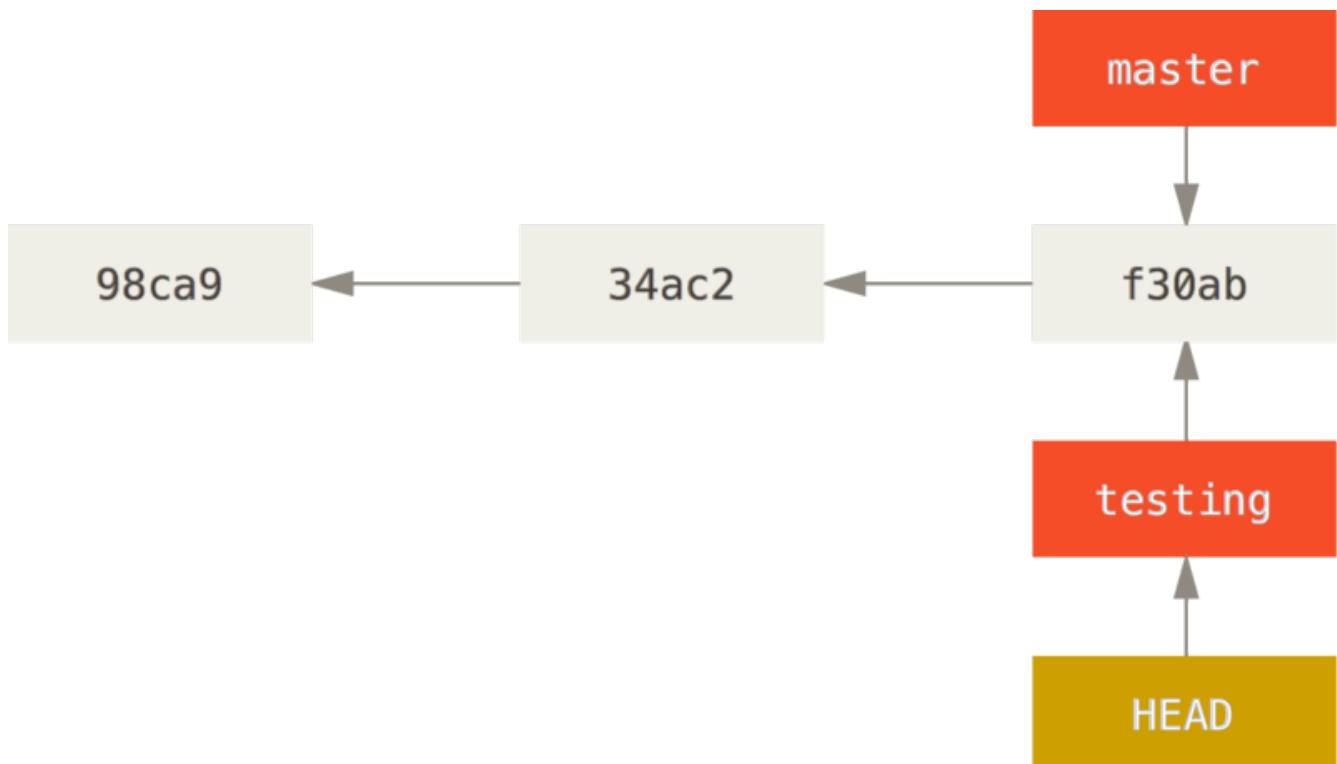


Figure 14. El apuntador HEAD apunta a la rama actual

¿Cuál es el significado de todo esto? Bueno... lo veremos tras realizar otra confirmación de cambios:

```
$ vim test.rb
$ git commit -a -m 'made a change'
```

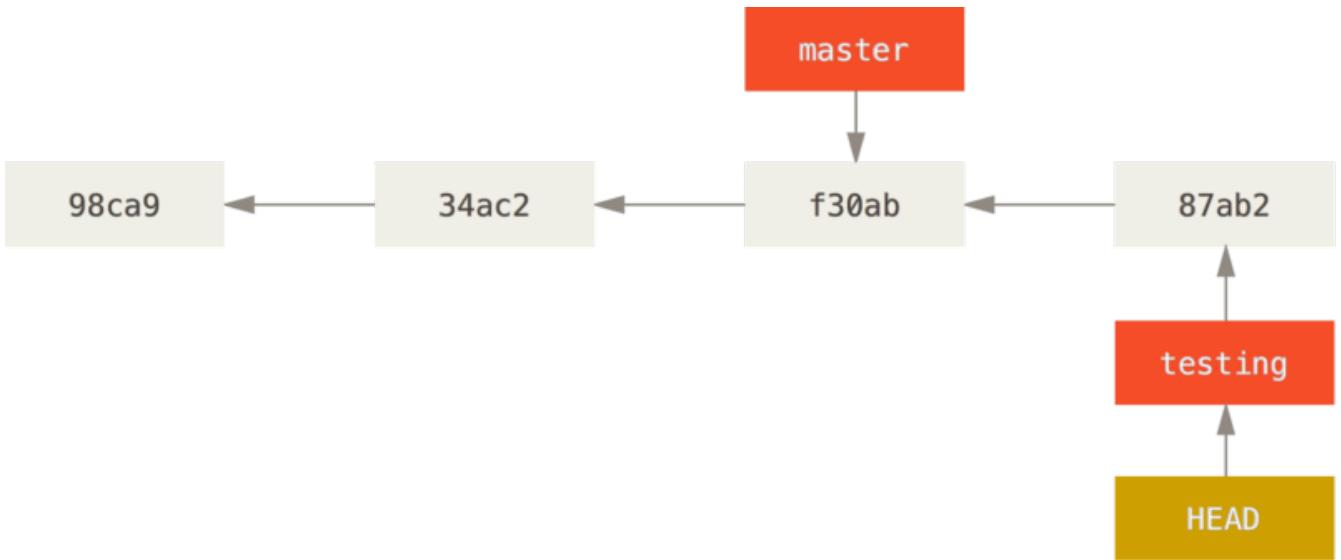


Figure 15. La rama apuntada por HEAD avanza con cada confirmación de cambios

Observamos algo interesante: la rama `testing` avanza, mientras que la rama `master` permanece en la confirmación donde estaba cuando lanzaste el comando `git checkout` para saltar. Volvamos ahora a la rama `master`:

```
$ git checkout master
```

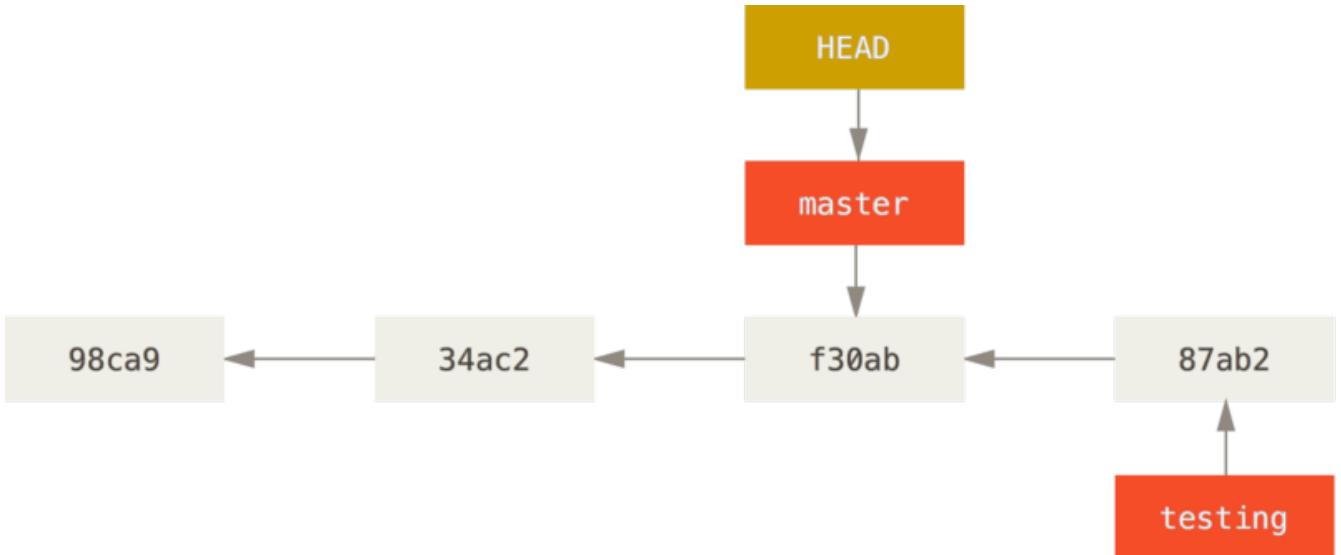


Figure 16. HEAD apunta a otra rama cuando hacemos un salto

Este comando realiza **dos acciones**: Mueve el apuntador `HEAD` de nuevo a la rama `master`, y **revierte los archivos de tu directorio de trabajo**; dejándolos tal y como estaban en la última instantánea confirmada en dicha rama `master`. Esto supone que los cambios que hagas desde este momento en adelante divergirán de la antigua versión del proyecto. Básicamente, lo que se está haciendo es **rebobinar** el trabajo que habías hecho temporalmente en la rama `testing`; de tal forma que puedas avanzar en otra dirección diferente.

### *Saltar entre ramas cambia archivos en tu directorio de trabajo*

#### NOTE

Es importante destacar que cuando saltas a una rama en Git, los archivos de tu directorio de trabajo cambian. Si saltas a una rama antigua, tu directorio de trabajo retrocederá para verse como lo hacía la última vez que confirmaste un cambio en dicha rama. Si Git no puede hacer el cambio limpiamente, no te dejará saltar.

Haz algunos cambios más y confírmalos:

```
$ vim test.rb
$ git commit -a -m 'made other changes'
```

Ahora el historial de tu proyecto diverge (ver [Los registros de las ramas divergen](#)). Has creado una rama y saltado a ella, has trabajado sobre ella; has vuelto a la rama original, y has trabajado también sobre ella. Los cambios realizados en ambas sesiones de trabajo están aislados en ramas independientes: puedes saltar libremente de una a otra según estimes oportuno. Y todo ello simplemente con tres comandos: `git branch`, `git checkout` y `git commit`.

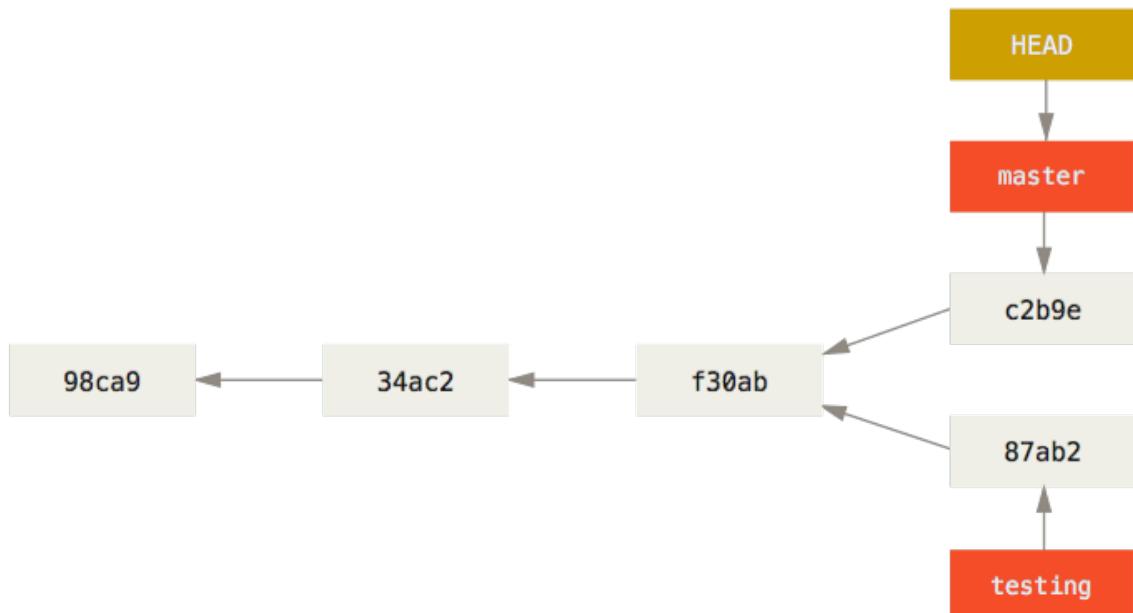


Figure 17. Los registros de las ramas divergen

También puedes ver esto fácilmente utilizando el comando `git log`. Si ejecutas `git log --oneline --decorate --graph --all` te mostrará el historial de tus confirmaciones, indicando dónde están los apuntadores de tus ramas y como ha divergido tu historial.

```
$ git log --oneline --decorate --graph --all
* c2b9e (HEAD, master) made other changes
| * 87ab2 (testing) made a change
|/
* f30ab add feature #32 - ability to add new formats to the
* 34ac2 fixed bug #1328 - stack overflow under certain conditions
* 98ca9 initial commit of my project
```

Debido a que una rama Git es realmente un simple archivo que contiene los 40 caracteres de una suma de control SHA-1, (representando la confirmación de cambios a la que apunta), no cuesta nada el crear y destruir ramas en Git. Crear una nueva rama es tan rápido y simple como escribir 41 bytes en un archivo, (40 caracteres y un retorno de carro).

Esto contrasta fuertemente con los métodos de ramificación usados por otros sistemas de control de versiones, en los que crear una rama nueva supone el copiar todos los archivos del proyecto a un directorio adicional nuevo. Esto puede llevar segundos o incluso minutos, dependiendo del tamaño del proyecto; mientras que en Git el proceso es siempre instantáneo. Y, además, debido a que se almacenan también los nodos padre para cada confirmación, el encontrar las bases adecuadas para realizar una fusión entre ramas es un proceso automático y generalmente sencillo de realizar. Animando así a los desarrolladores a utilizar ramificaciones frecuentemente.

Vamos a ver el por qué merece la pena hacerlo así.

## Procedimientos Básicos para Ramificar y Fusionar

Vamos a presentar un ejemplo simple de ramificar y de fusionar, con un flujo de trabajo que se podría presentar en la realidad. Imagina que sigues los siguientes pasos:

1. Trabajas en un sitio web.
2. Creas una rama para un nuevo tema sobre el que quieres trabajar.
3. Realizas algo de trabajo en esa rama.

En este momento, recibes una llamada avisándote de un problema crítico que has de resolver. Y sigues los siguientes pasos:

1. Vuelves a la rama de producción original.
2. Creas una nueva rama para el problema crítico y lo resuelves trabajando en ella.
3. Tras las pertinentes pruebas, fusionas (merge) esa rama y la envías (push) a la rama de producción.
4. Vuelves a la rama del tema en que andabas antes de la llamada y continuas tu trabajo.

## Procedimientos Básicos de Ramificación

Imagina que estas trabajando en un proyecto y tienes un par de confirmaciones (commit) ya realizadas.

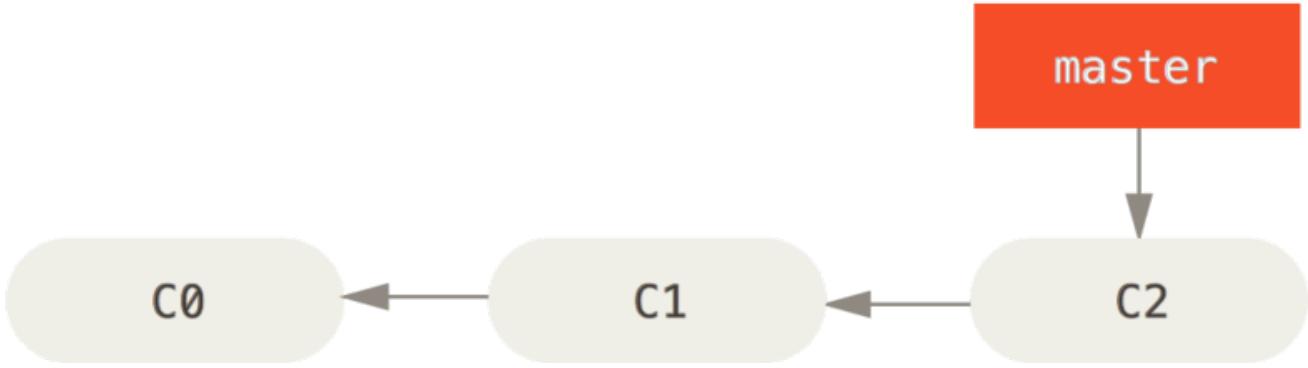


Figure 18. Un registro de confirmaciones corto y sencillo

Decides trabajar en el problema #53, según el sistema que tu compañía utiliza para llevar seguimiento de los problemas. Para crear una nueva rama y saltar a ella, en un solo paso, puedes utilizar el comando `git checkout` con la opción `-b`:

```
$ git checkout -b iss53
Switched to a new branch "iss53"
```

Esto es un **atajo** a:

```
$ git branch iss53
$ git checkout iss53
```

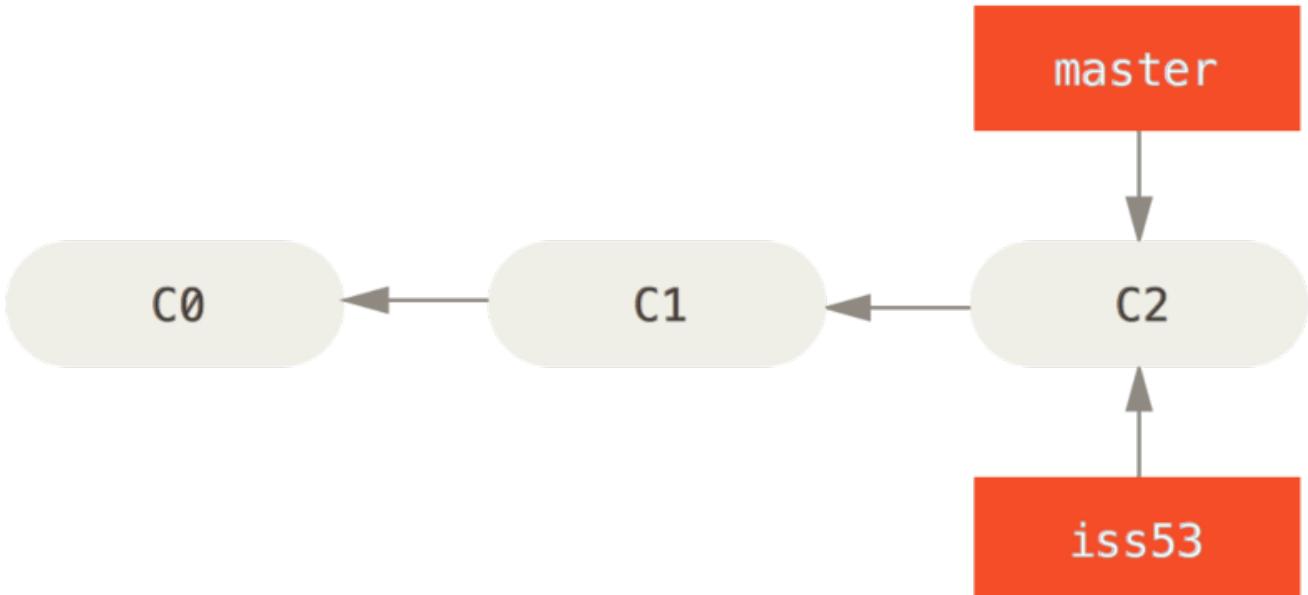


Figure 19. Crear un apuntador a la rama nueva

Trabajas en el sitio web y haces algunas confirmaciones de cambios (commits). Con ello avanzas la rama **iss53**, que es la que tienes activada (**checked out**) en este momento (es decir, a la que apunta HEAD):

```
$ vim index.html  
$ git commit -a -m 'added a new footer [issue 53]'
```

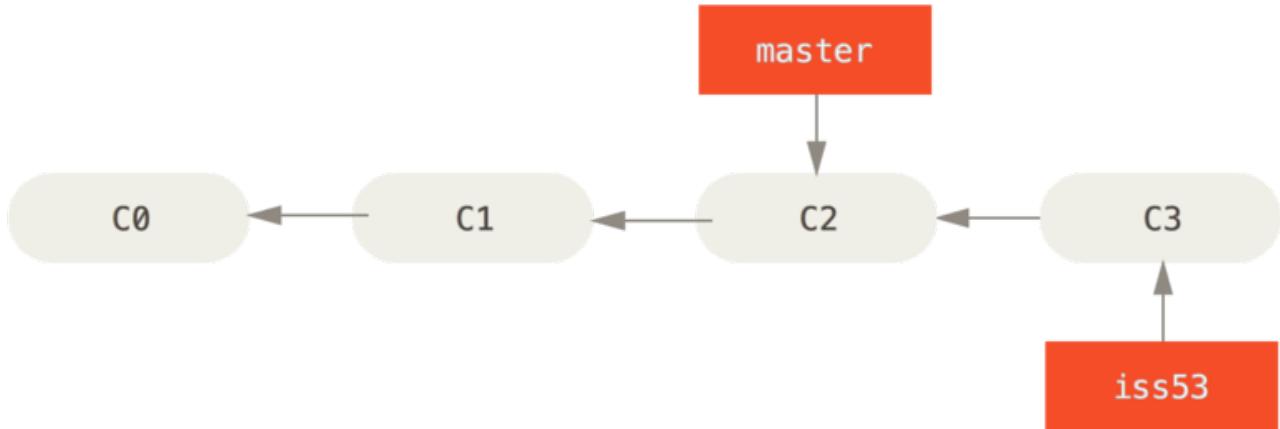


Figure 20. La rama `iss53` ha avanzado con tu trabajo

Entonces, recibes una llamada avisándote de otro problema urgente en el sitio web y debes resolverlo inmediatamente. Al usar Git, no necesitas mezclar el nuevo problema con los cambios que ya habías realizado sobre el problema #53; ni tampoco perder tiempo revirtiendo esos cambios para poder trabajar sobre el contenido que está en producción. Basta con saltar de nuevo a la rama `master` y continuar trabajando a partir de allí.

Pero, antes de poder hacer eso, hemos de tener en cuenta que si tenemos cambios aún no confirmados en el directorio de trabajo o en el área de preparación, Git no nos permitirá saltar a otra rama con la que podríamos tener conflictos. Lo mejor es tener siempre un estado de trabajo limpio y despejado antes de saltar entre ramas. Y, para ello, tenemos algunos procedimientos (`stash` y corregir confirmaciones), que vamos a ver más adelante en [Guardado rápido y Limpieza](#). Por ahora, como tenemos confirmados todos los cambios, podemos saltar a la rama `master` sin problemas:

```
$ git checkout master  
Switched to branch 'master'
```

Tras esto, tendrás el directorio de trabajo exactamente igual a como estaba antes de comenzar a trabajar sobre el problema #53 y podrás concentrarte en el nuevo problema urgente. Es importante recordar que Git revierte el directorio de trabajo exactamente al estado en que estaba en la confirmación (commit) apuntada por la rama que activamos (`checkout`) en cada momento. Git añade, quita y modifica archivos automáticamente para asegurar que tu copia de trabajo luce exactamente como lucía la rama en la última confirmación de cambios realizada sobre ella.

A continuación, es momento de resolver el problema urgente. Vamos a crear una nueva rama `hotfix`, sobre la que trabajar hasta resolverlo:

```
$ git checkout -b hotfix
Switched to a new branch 'hotfix'
$ vim index.html
$ git commit -a -m 'fixed the broken email address'
[hotfix 1fb7853] fixed the broken email address
 1 file changed, 2 insertions(+)
```

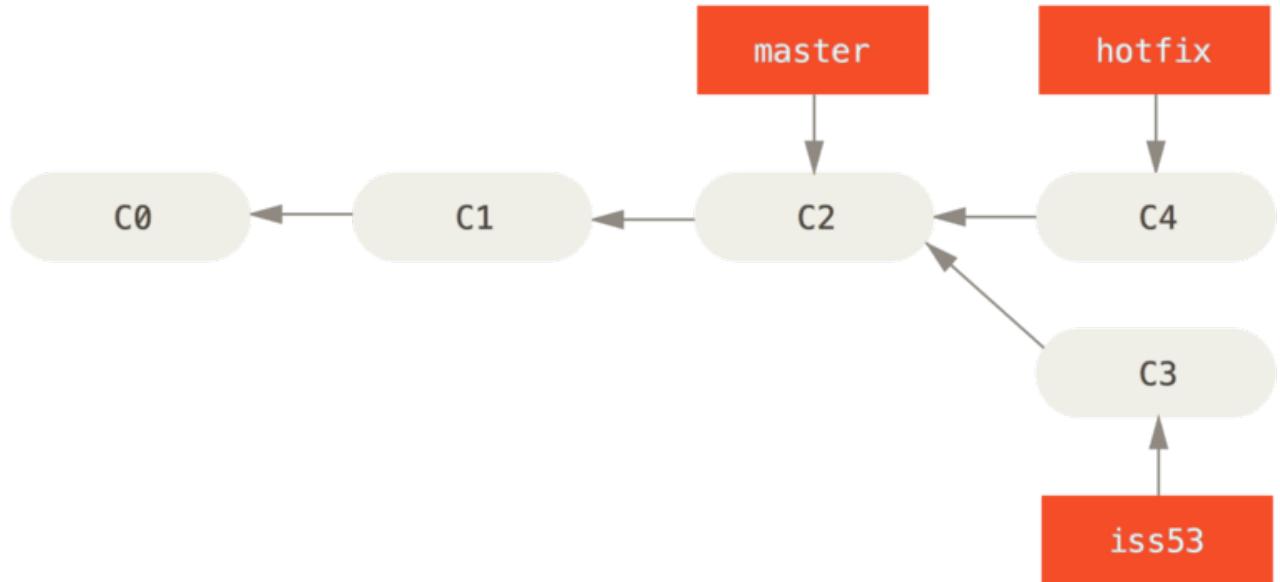


Figure 21. Rama `hotfix` basada en la rama `master` original

Puedes realizar las pruebas oportunas, asegurarte que la solución es correcta, e incorporar los cambios a la rama `master` para ponerlos en producción. Esto se hace con el comando `git merge`:

```
$ git checkout master
$ git merge hotfix
Updating f42c576..3a0874c
Fast-forward
 index.html | 2 ++
 1 file changed, 2 insertions(+)
```

Notarás la frase “Fast forward” (“Avance rápido”, en inglés) que aparece en la salida del comando. Git ha movido el apuntador hacia adelante, ya que la confirmación apuntada en la rama donde has fusionado estaba directamente arriba respecto a la confirmación actual. Dicho de otro modo: cuando intentas fusionar una confirmación con otra confirmación accesible siguiendo directamente el historial de la primera; Git simplifica las cosas avanzando el puntero, ya que no hay ningún otro trabajo divergente a fusionar. Esto es lo que se denomina “avance rápido” (“fast forward”).

Ahora, los cambios realizados están ya en la instantánea (snapshot) de la confirmación (commit) apuntada por la rama `master`. Y puedes desplegarlos.

Figure 22. Tras la fusión (merge), la rama `master` apunta al mismo sitio que la rama `hotfix`.

Tras haber resuelto el problema urgente que había interrumpido tu trabajo, puedes volver a donde estabas. Pero antes, es importante borrar la rama `hotfix`, ya que no la vamos a necesitar más, puesto que apunta exactamente al mismo sitio que la rama `master`. Esto lo puedes hacer con la opción `-d` del comando `git branch`:

```
$ git branch -d hotfix  
Deleted branch hotfix (3a0874c).
```

Y, con esto, ya estás listo para regresar al trabajo sobre el problema #53.

```
$ git checkout iss53  
Switched to branch "iss53"  
$ vim index.html  
$ git commit -a -m 'finished the new footer [issue 53]'  
[iss53 ad82d7a] finished the new footer [issue 53]  
1 file changed, 1 insertion(+)
```

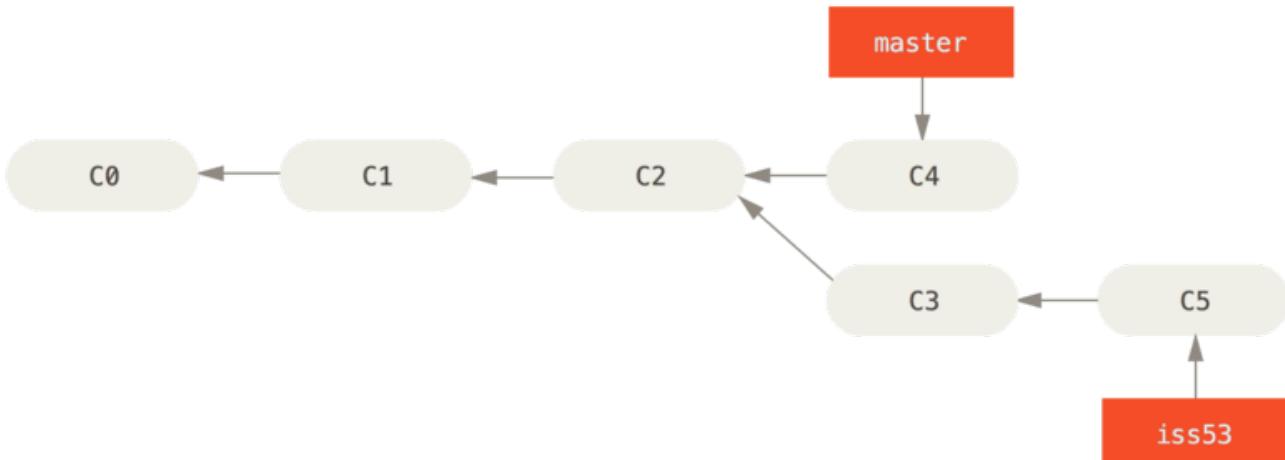


Figure 23. La rama `iss53` puede avanzar independientemente

Cabe destacar que todo el trabajo realizado en la rama `hotfix` no está en los archivos de la rama `iss53`. Si fuera necesario agregarlos, puedes fusionar (merge) la rama `master` sobre la rama `iss53` utilizando el comando `git merge master`, o puedes esperar hasta que decidas fusionar (merge) la rama `iss53` a la rama `master`.

## Procedimientos Básicos de Fusión

Supongamos que tu trabajo con el problema #53 ya está completo y listo para fusionarlo (merge) con la rama `master`. Para ello, de forma similar a como antes has hecho con la rama `hotfix`, vas a fusionar la rama `iss53`. Simplemente, activa (checkout) la rama donde deseas fusionar y lanza el comando `git merge`:

```
$ git checkout master
Switched to branch 'master'
$ git merge iss53
Merge made by the 'recursive' strategy.
index.html |    1 +
1 file changed, 1 insertion(+)
```

Es algo diferente de la fusión realizada anteriormente con `hotfix`. En este caso, el registro de desarrollo había divergido en un punto anterior. Debido a que la confirmación en la rama actual no es ancestro directo de la rama que pretendes fusionar, Git tiene cierto trabajo extra que hacer. Git realizará una fusión a tres bandas, utilizando las dos instantáneas apuntadas por el extremo de cada una de las ramas y por el ancestro común a ambas.

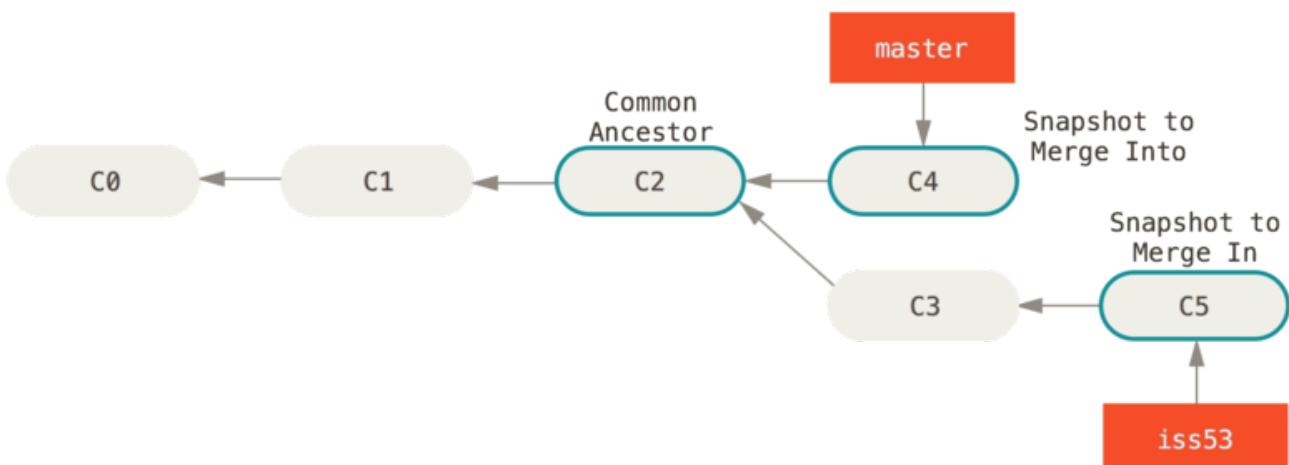


Figure 24. Git identifica automáticamente el mejor ancestro común para realizar la fusión de las ramas

En lugar de simplemente avanzar el apuntador de la rama, Git crea una nueva instantánea (snapshot) resultante de la fusión a tres bandas; y crea automáticamente una nueva confirmación de cambios (commit) que apunta a ella. Nos referimos a este proceso como "fusión confirmada" y su particularidad es que tiene más de un parent.

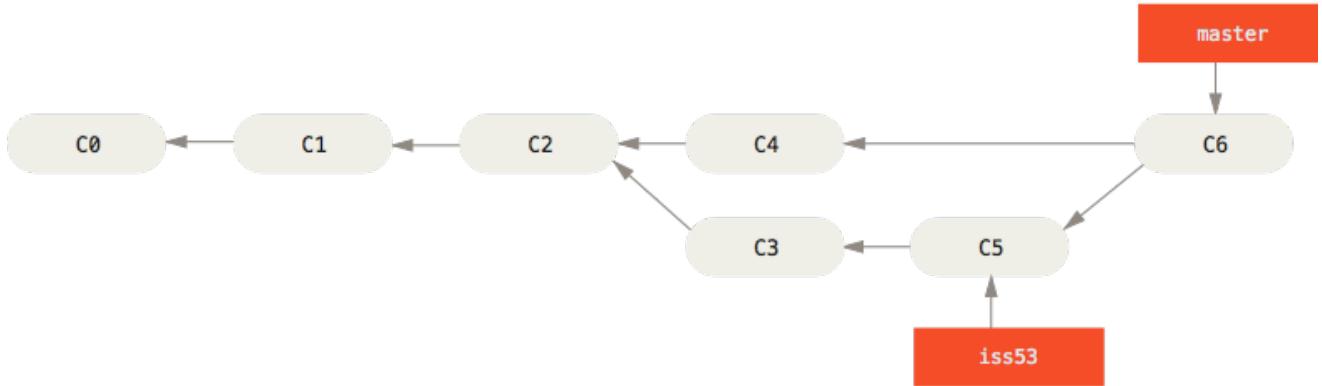


Figure 25. Git crea automáticamente una nueva confirmación para la fusión

Vale la pena destacar el hecho de que es el propio Git quien determina automáticamente el mejor ancestro común para realizar la fusión; a diferencia de otros sistemas tales como CVS o Subversion, donde es el desarrollador quien ha de determinar cuál puede ser dicho mejor ancestro común. Esto hace que en Git sea mucho más fácil realizar fusiones.

Ahora que todo tu trabajo ya está fusionado con la rama principal, no tienes necesidad de la rama iss53. Por lo que puedes borrarla y cerrar manualmente el problema en el sistema de seguimiento de problemas de tu empresa.

```
$ git branch -d iss53
```

## Principales Conflictos que Pueden Surgir en las Fusiones

En algunas ocasiones, los procesos de fusión no suelen ser fluidos. Si hay modificaciones dispares en una misma porción de un mismo archivo en las dos ramas distintas que pretendes fusionar, Git no será capaz de fusionarlas directamente. Por ejemplo, si en tu trabajo del problema #53 has modificado una misma porción que también ha sido modificada en el problema **hotfix**, verás un conflicto como este:

```
$ git merge iss53
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

Git no crea automáticamente una nueva fusión confirmada (merge commit), sino que **hace una pausa en el proceso**, esperando a que tú resuelvas el conflicto. Para ver qué archivos permanecen sin fusionar en un determinado momento conflictivo de una fusión, puedes usar el comando **git status**:

```
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

    both modified:    index.html

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

Todo aquello que sea conflictivo y no se haya podido resolver, se marca como "sin fusionar" (unmerged). Git añade a los archivos conflictivos unos marcadores especiales de resolución de conflictos que te guiarán cuando abras manualmente los archivos implicados y los edites para corregirlos. El archivo conflictivo contendrá algo como:

```
<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=====
<div id="footer">
  please contact us at support@github.com
</div>
>>>>> iss53:index.html
```

Donde nos dice que la versión en HEAD (la rama **master**, la que habías activado antes de lanzar el comando de fusión) contiene lo indicado **en la parte superior del bloque** (todo lo que está encima de **=====**) y que la versión en **iss53** contiene el resto, lo indicado **en la parte inferior del bloque**. Para resolver el conflicto, **has de elegir manualmente** el contenido de uno o de otro lado. Por ejemplo, puedes optar por cambiar el bloque, dejándolo así:

```
<div id="footer">
  please contact us at email.support@github.com
</div>
```

Esta corrección contiene un poco de ambas partes y se han eliminado completamente las líneas **<<<<<**, **=====** y **>>>>>**. Tras resolver todos los bloques conflictivos, has de lanzar comandos **git add** para marcar cada archivo modificado. Marcar archivos como preparados (staged) indica a Git que sus conflictos han sido resueltos.

Si en lugar de resolver directamente prefieres utilizar una herramienta gráfica, puedes usar el comando **git mergetool**, el cual arrancará la correspondiente herramienta de visualización y te permitirá ir resolviendo conflictos con ella:

```
$ git mergetool

This message is displayed because 'merge.tool' is not configured.
See 'git mergetool --tool-help' or 'git help config' for more details.
'git mergetool' will now attempt to use one of the following tools:
opendiff kdiff3 tkdiff xxdiff meld tortoisemerge gvimdiff diffuse diffmerge ecmerge
p4merge araxis bc3 codecompare vimdiff emerge
Merging:
index.html

Normal merge conflict for 'index.html':
{local}: modified file
{remote}: modified file
Hit return to start merge resolution tool (opendiff):
```

Si deseas usar una herramienta distinta de la escogida por defecto (en mi caso `opendiff`, porque estoy lanzando el comando en Mac), puedes escogerla entre la lista de herramientas soportadas mostradas al principio ("merge tool candidates") tecleando el nombre de dicha herramienta.

**NOTE**

Si necesitas herramientas más avanzadas para resolver conflictos de fusión más complicados, revisa la sección de fusionado en [Fusión Avanzada](#).

Tras salir de la herramienta de fusionado, Git preguntará si hemos resuelto todos los conflictos y la fusión ha sido satisfactoria. Si le indicas que así ha sido, Git marca como preparado (staged) el archivo que acabamos de modificar. En cualquier momento, puedes lanzar el comando `git status` para ver si ya has resuelto todos los conflictos:

```
$ git status
On branch master
All conflicts fixed but you are still merging.
(use "git commit" to conclude merge)

Changes to be committed:

modified:   index.html
```

Si todo ha ido correctamente, y ves que todos los archivos conflictivos están marcados como preparados, puedes lanzar el comando `git commit` para terminar de confirmar la fusión. El mensaje de confirmación por defecto será algo parecido a:

```

Merge branch 'iss53'

Conflicts:
  index.html
#
# It looks like you may be committing a merge.
# If this is not correct, please remove the file
#   .git/MERGE_HEAD
# and try again.

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# All conflicts fixed but you are still merging.
#
# Changes to be committed:
#   modified:   index.html
#

```

Puedes modificar este mensaje añadiendo detalles sobre cómo has resuelto la fusión, si lo consideras útil para que otros entiendan esta fusión en un futuro. Se trata de indicar por qué has hecho lo que has hecho; a no ser que resulte obvio, claro está.

## Gestión de Ramas

Ahora que ya has creado, fusionado y borrado algunas ramas, vamos a dar un vistazo a algunas herramientas de gestión muy útiles cuando comienzas a utilizar ramas de manera avanzada.

El comando `git branch` tiene más funciones que las de crear y borrar ramas. Si lo lanzas sin parámetros, obtienes una lista de las ramas presentes en tu proyecto:

```

$ git branch
  iss53
* master
  testing

```

Fíjate en el carácter `*` delante de la rama `master`: nos indica la rama activa en este momento (la rama a la que apunta `HEAD`). Si hacemos una confirmación de cambios (`commit`), esa será la rama que avance. Para ver la última confirmación de cambios en cada rama, puedes usar el comando `git branch -v`:

```

$ git branch -v
  iss53  93b412c fix javascript issue
* master  7a98805 Merge branch 'iss53'
  testing 782fd34 add scott to the author list in the readmes

```

Otra opción útil para averiguar el estado de las ramas, es filtrarlas y mostrar solo aquellas que han sido fusionadas (o que no lo han sido) con la rama actualmente activa. Para ello, Git dispone de las opciones `--merged` y `--no-merged`. Si deseas ver las ramas que han sido fusionadas en la rama activa, puedes lanzar el comando `git branch --merged`:

```
$ git branch --merged  
iss53  
* master
```

Aparece la rama `iss53` porque ya ha sido fusionada. Las ramas que no llevan por delante el carácter `*` pueden ser eliminadas sin problemas, porque todo su contenido ya ha sido incorporado a otras ramas.

Para mostrar todas las ramas que contienen trabajos sin fusionar, puedes utilizar el comando `git branch --no-merged`:

```
$ git branch --no-merged  
testing
```

Esto nos muestra la otra rama del proyecto. Debido a que contiene trabajos sin fusionar, al intentarla borrarla con `git branch -d`, el comando nos dará un error:

```
$ git branch -d testing  
error: The branch 'testing' is not fully merged.  
If you are sure you want to delete it, run 'git branch -D testing'.
```

Si realmente deseas borrar la rama, y perder el trabajo contenido en ella, puedes forzar el borrado con la opción `-D`; tal y como indica el mensaje de ayuda.

## Flujos de Trabajo Ramificados

Ahora que ya has visto los procedimientos básicos de ramificación y fusión, ¿qué puedes o qué debes hacer con ellos? En este apartado vamos a ver algunos de los flujos de trabajo más comunes, de tal forma que puedas decidir si te gustaría incorporar alguno de ellos a tu ciclo de desarrollo.

### Ramas de Largo Recorrido

Por la sencillez de la fusión a tres bandas de Git, el fusionar una rama a otra varias veces a lo largo del tiempo es fácil de hacer. Esto te posibilita tener varias ramas siempre abiertas, e ir las usando en diferentes etapas del ciclo de desarrollo; realizando fusiones frecuentes entre ellas.

Muchos desarrolladores que usan Git llevan un flujo de trabajo de esta naturaleza, manteniendo en la rama `master` únicamente el código totalmente `estable` (el código que ha sido o que va a ser liberado) y teniendo otras ramas paralelas denominadas `desarrollo` o `siguiente`, en las que trabajan y realizan pruebas. Estas ramas paralelas no suele estar siempre en un estado estable; pero cada vez que sí lo están, pueden ser fusionadas con la rama `master`. También es habitual el incorporarle

(pull) ramas puntuales (ramas temporales, como la rama `iss53` del ejemplo anterior) cuando las completamos y estamos seguros de que no van a introducir errores.

En realidad, en todo momento estamos hablando simplemente de apuntadores moviéndose por la línea temporal de confirmaciones de cambio (commit history). Las ramas estables apuntan hacia posiciones más antiguas en el historial de confirmaciones, mientras que las ramas avanzadas, las que van abriendo camino, apuntan hacia posiciones más recientes.

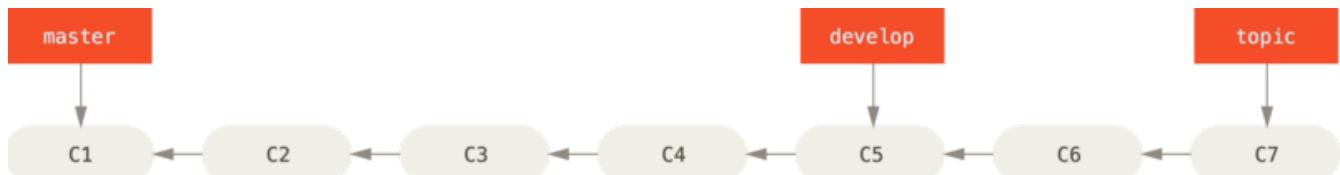


Figure 26. Una vista lineal del ramificado progresivo estable

Podría ser más sencillo pensar en las ramas como si fueran silos de almacenamiento, donde grupos de confirmaciones de cambio (commits) van siendo promocionados hacia silos más estables a medida que son probados y depurados.

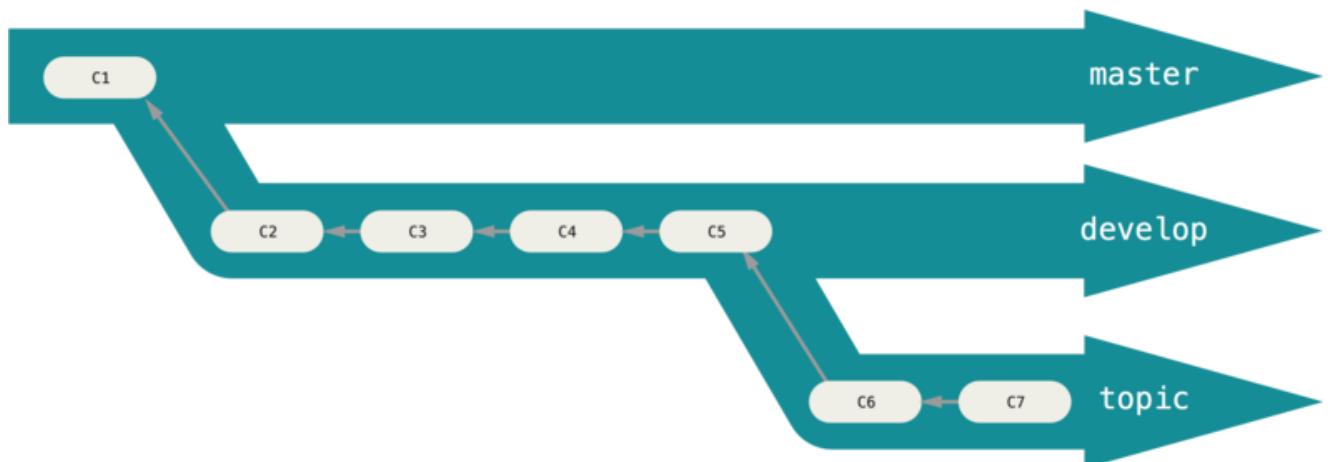


Figure 27. Una vista tipo “silo” del ramificado progresivo estable

Este sistema de trabajo se puede ampliar para diversos grados de estabilidad. Algunos proyectos muy grandes suelen tener una rama denominada `propuestas` o `pu` (del inglés “proposed updates”, propuesta de actualización), donde suele estar todo aquello integrado desde otras ramas, pero que aún no está listo para ser incorporado a las ramas `siguiente` o `master`. La idea es mantener siempre diversas ramas en diversos grados de estabilidad; pero cuando alguna alcanza un estado más estable, la fusionamos con la rama inmediatamente superior a ella. Aunque no es obligatorio el trabajar con ramas de larga duración, realmente es práctico y útil, sobre todo en proyectos largos o complejos.

## Ramas Puntuales

Las ramas puntuales, en cambio, son útiles en proyectos de cualquier tamaño. Una rama puntual es aquella rama de corta duración que abre para un tema o para una funcionalidad determinada. Es algo que nunca habrías hecho en otro sistema VCS, debido a los altos costos de crear y fusionar ramas en esos sistemas. Pero en Git, por el contrario, es muy habitual el crear, trabajar con, fusionar y eliminar ramas varias veces al día.

Tal y como has visto con las ramas `iss53` y `hotfix` que has creado en la sección anterior. Has hecho algunas confirmaciones de cambio en ellas, y luego las has borrado tras fusionarlas con la rama principal. Esta técnica te posibilita realizar cambios de contexto rápidos y completos y, debido a que el trabajo está claramente separado en silos, con todos los cambios de cada tema en su propia rama, te será mucho más sencillo revisar el código y seguir su evolución. Puedes mantener los cambios ahí durante minutos, días o meses; y fusionarlos cuando realmente estén listos, sin importar el orden en el que fueron creados o en el que comenzaste a trabajar en ellos.

Por ejemplo, puedes realizar cierto trabajo en la rama `master`, ramificar para un problema concreto (rama `iss91`), trabajar en él un rato, ramificar una segunda vez para probar otra manera de resolverlo (rama `iss92v2`), volver a la rama `master` y trabajar un poco más, y, por último, ramificar temporalmente para probar algo de lo que no estás seguro (rama `dumbidea`). El historial de confirmaciones (commit history) será algo parecido esto:

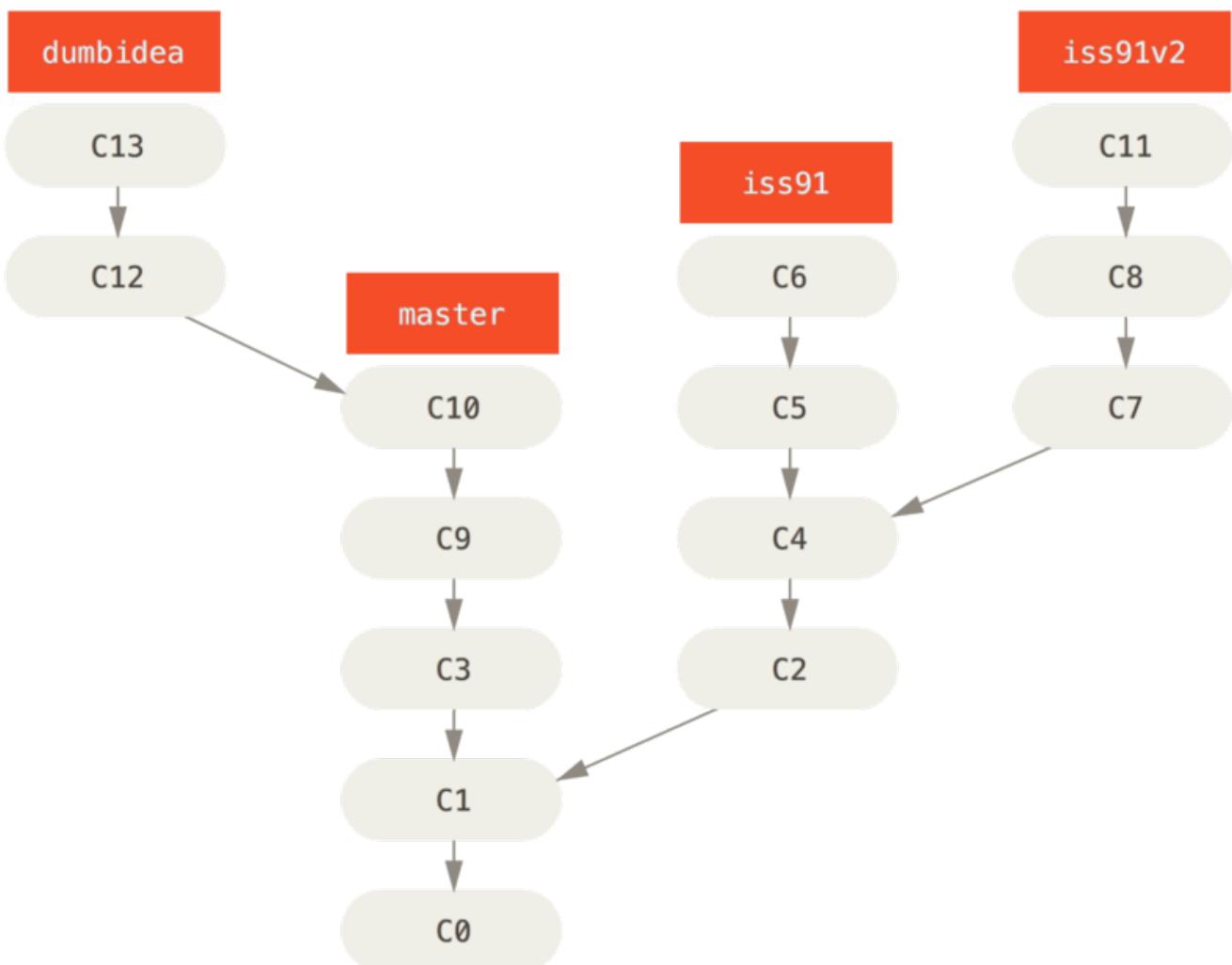


Figure 28. Múltiples ramas puntuales

En este momento, supongamos que te decides por la segunda solución al problema (rama `iss92v2`); y que, tras mostrar la rama `dumbidea` a tus compañeros, resulta que les parece una idea genial. Puedes descartar la rama `iss91` (perdiendo las confirmaciones `C5` y `C6`), y fusionar las otras dos. El historial será algo parecido a esto:

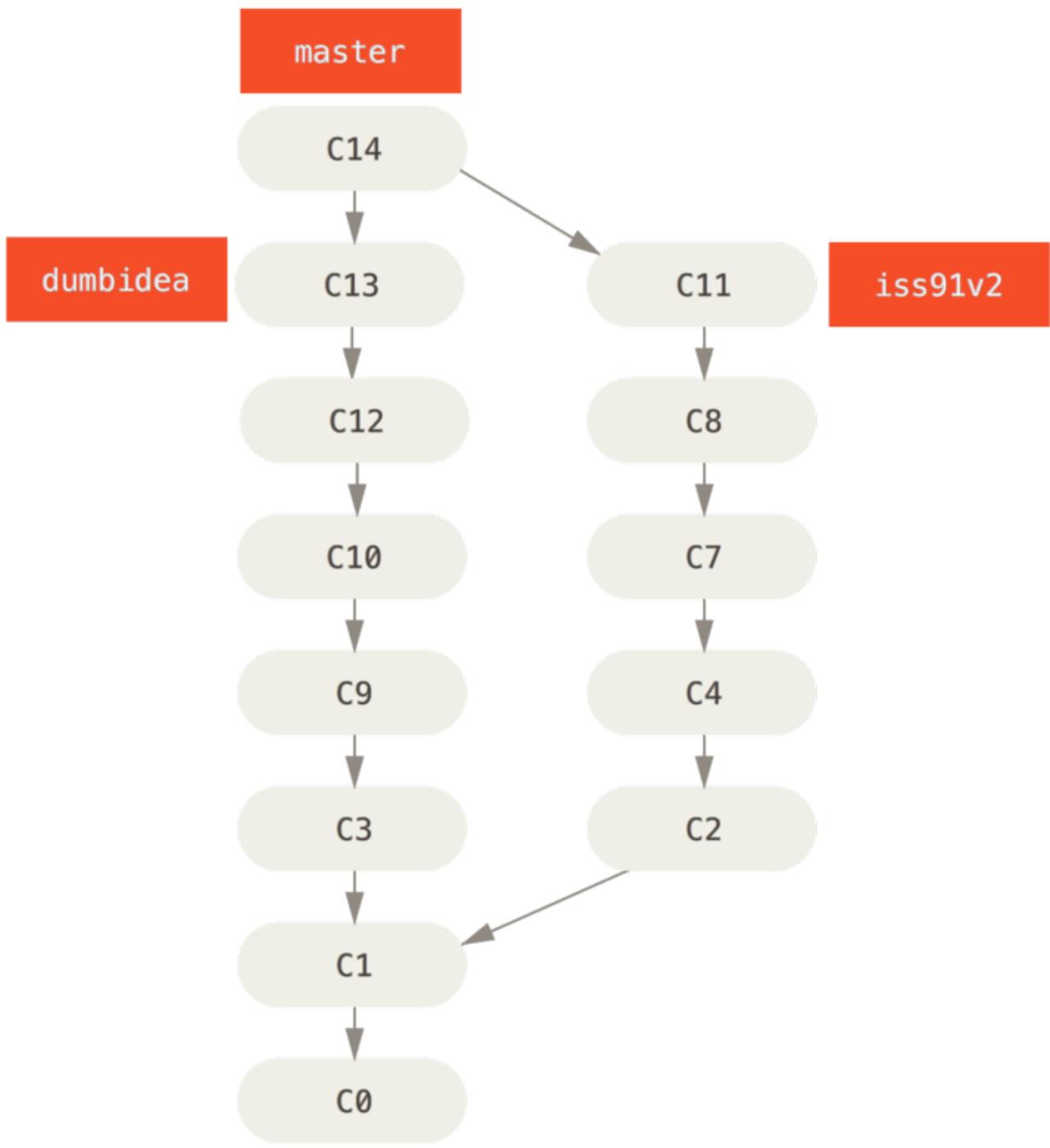


Figure 29. El historial tras fusionar `dumbidea` e `iss91v2`

Hablaremos un poco más sobre los distintos flujos de trabajo de tu proyecto Git en [Git en entornos distribuidos](#), así que antes de decidir qué estilo de ramificación usará tu próximo proyecto, asegúrate de haber leído ese capítulo.

Es importante recordar que, mientras estás haciendo todo esto, todas las ramas son completamente locales. [Cuando ramificas y fusionas, todo se realiza en tu propio repositorio Git](#). No hay ningún tipo de comunicación con ningún servidor.

## Ramas Remotas

Las ramas remotas son referencias al estado de las ramas en tus repositorios remotos. [Son ramas](#)

locales que no puedes mover; se mueven automáticamente cuando estableces comunicaciones en la red. Las ramas remotas funcionan como marcadores, para recordarte en qué estado se encontraban tus repositorios remotos la última vez que conectaste con ellos.

Suelen referenciarse como `(remoto)/(rama)`. Por ejemplo, si quieres saber cómo estaba la rama `master` en el remoto `origin`, puedes revisar la rama `origin/master`. O si estás trabajando en un problema con un compañero y este envía (push) una rama `iss53`, tú tendrás tu propia rama de trabajo local `iss53`; pero la rama en el servidor apuntará a la última confirmación (commit) en la rama `origin/iss53`.

Esto puede ser un tanto confuso, pero intentemos aclararlo con un ejemplo. Supongamos que tienes un servidor Git en tu red, en `git.ourcompany.com`. Si haces un clon desde ahí, Git automáticamente lo denominará `origin`, traerá (pull) sus datos, creará un apuntador hacia donde esté en ese momento su rama `master` y denominará la copia local `origin/master`. Git te proporcionará también tu propia rama `master`, apuntando al mismo lugar que la rama `master` de `origin`; de manera que tengas donde trabajar.

#### *“origin” no es especial*

##### NOTE

Así como la rama “`master`” no tiene ningún significado especial en Git, tampoco lo tiene “`origin`”. “`master`” es un nombre muy usado solo porque es el nombre por defecto que Git le da a la rama inicial cuando ejecutas `git init`. De la misma manera, “`origin`” es el nombre por defecto que Git le da a un remoto cuando ejecutas `git clone`. Si en cambio ejecutases `git clone -o booyah`, tendrías una rama `booyah/master` como rama remota por defecto.

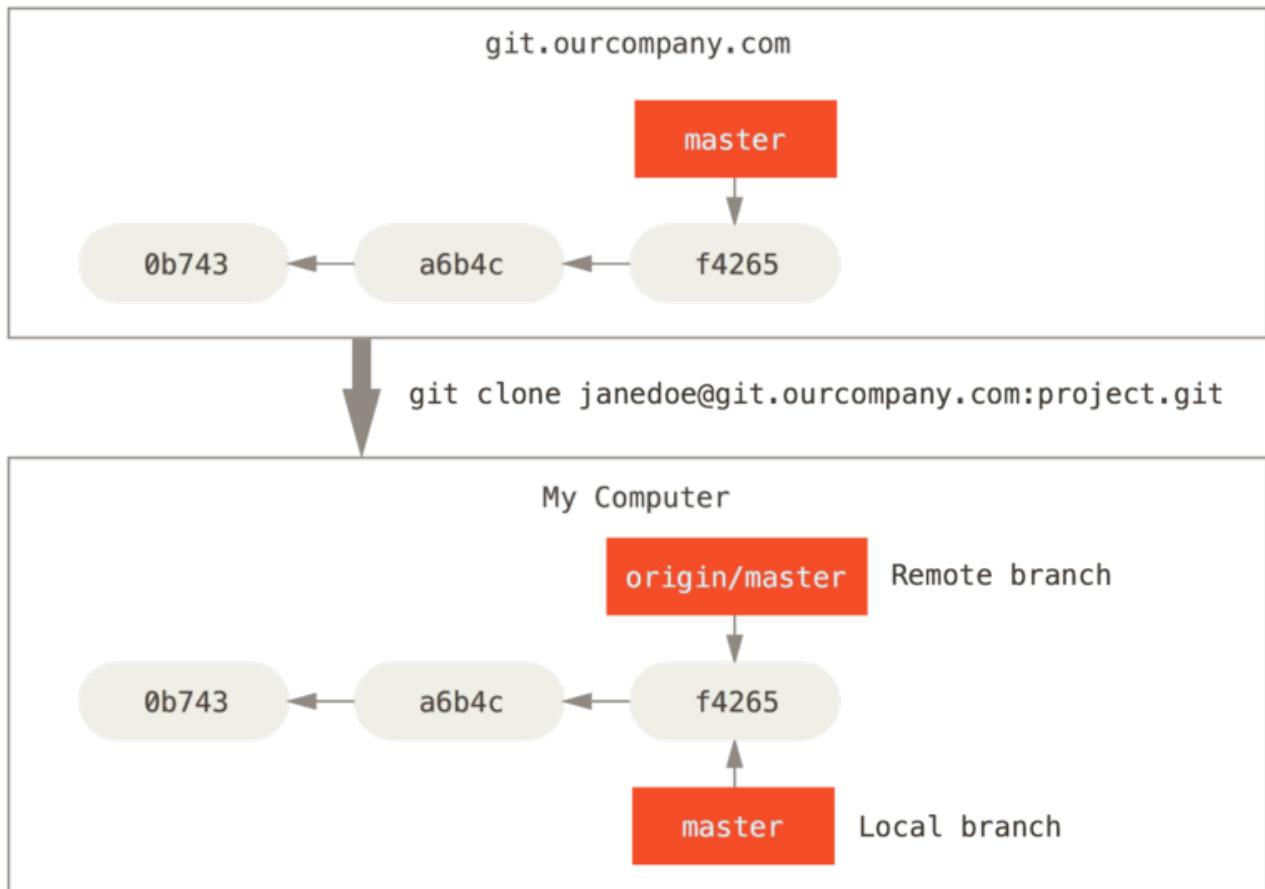


Figure 30. Servidor y repositorio local luego de ser clonado

Si haces algún trabajo en tu rama `master` local, y al mismo tiempo, alguien más lleva (push) su trabajo al servidor `git.ourcompany.com`, actualizando la rama `master` de allí, te encontrarás con que ambos registros avanzan de forma diferente. Además, mientras no tengas contacto con el servidor, tu apuntador a tu rama `origin/master` no se moverá.

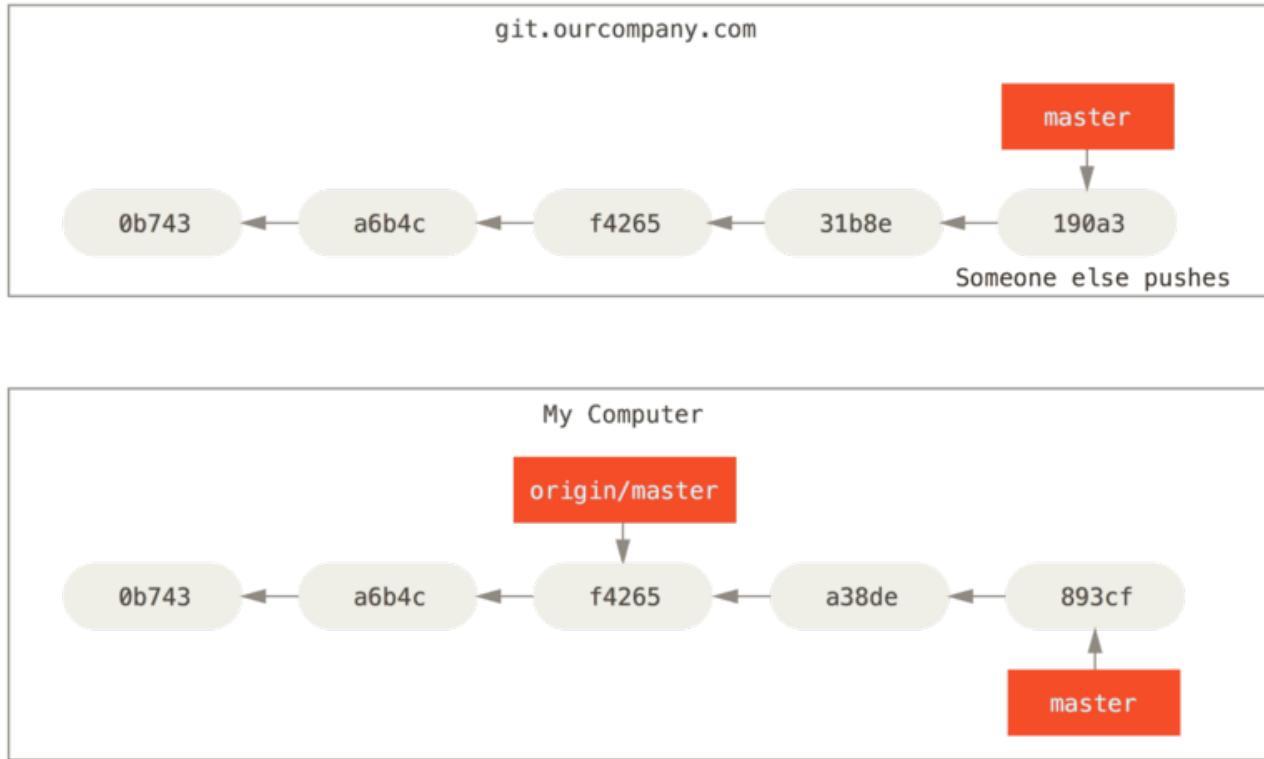


Figure 31. El trabajo remoto y el local pueden diverger

Para sincronizarte, puedes utilizar el comando `git fetch origin`. Este comando localiza en qué servidor está el origen (en este caso `git.ourcompany.com`), recupera cualquier dato presente allí que tú no tengas, y actualiza tu base de datos local, moviendo tu rama `origin/master` para que apunte a la posición más reciente.

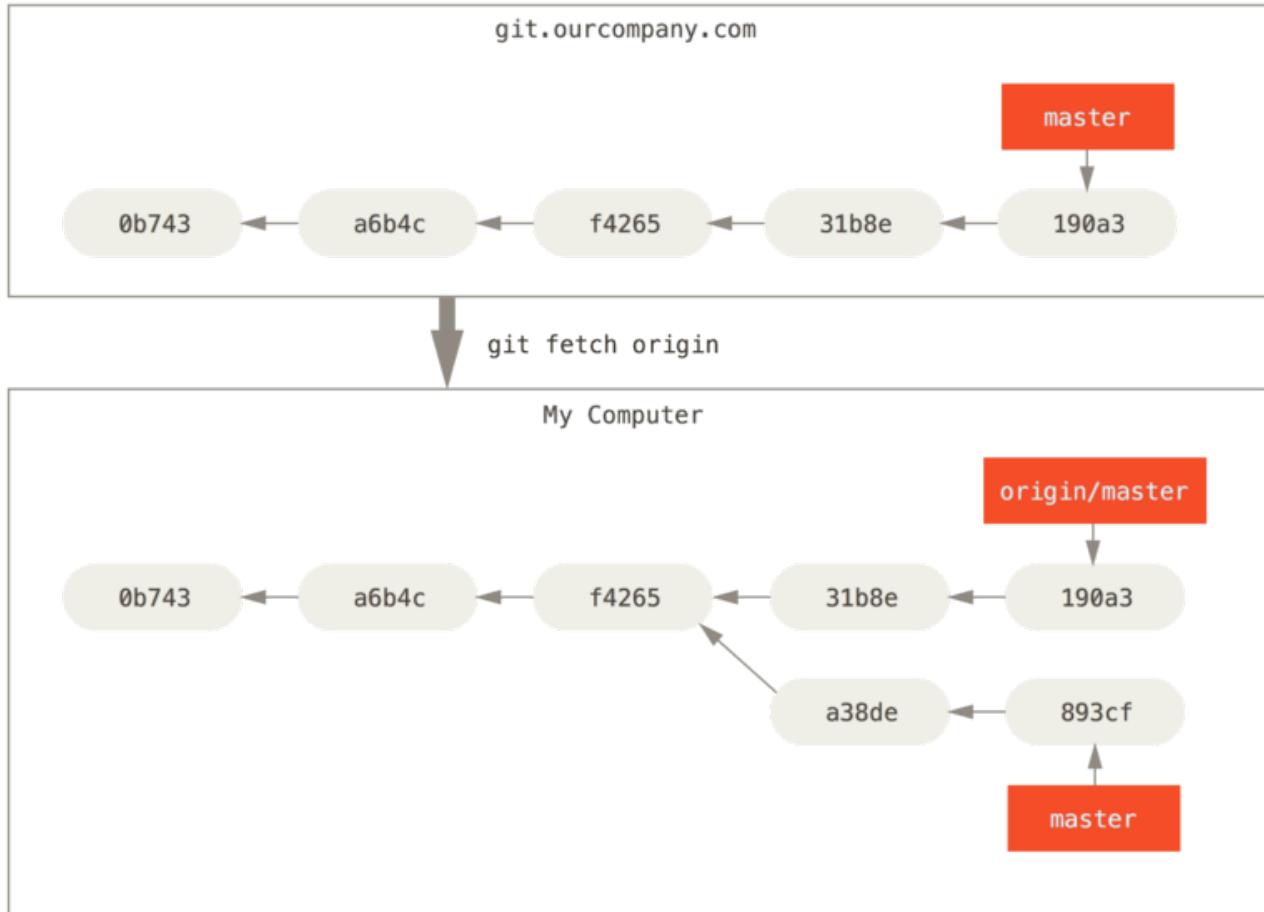


Figure 32. `git fetch` actualiza las referencias de tu remoto

Para ilustrar mejor el caso de tener múltiples servidores y cómo van las ramas remotas para esos proyectos remotos, supongamos que tienes otro servidor Git; utilizado por uno de tus equipos sprint, solamente para desarrollo. Este servidor se encuentra en `git.team1.ourcompany.com`. Puedes incluirlo como una nueva referencia remota a tu proyecto actual, mediante el comando `git remote add`, tal y como vimos en [Fundamentos de Git](#). Puedes denominar `teamone` a este remoto al asignarle este nombre a la URL.

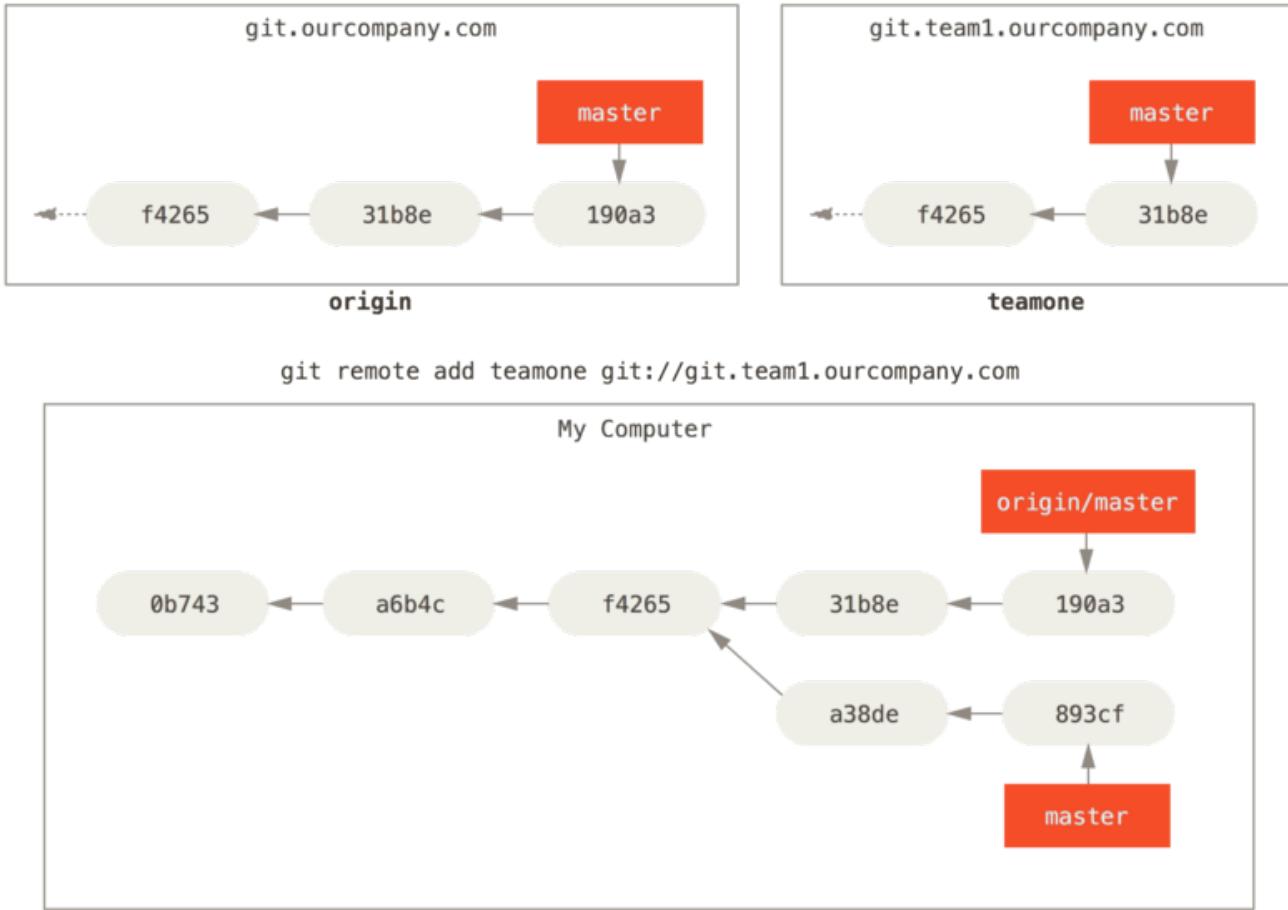


Figure 33. Añadiendo otro servidor como remoto

Ahora, puedes usar el comando `git fetch teamone` para recuperar todo el contenido del remoto `teamone` que tú no tenías. Debido a que dicho servidor es un subconjunto de los datos del servidor `origin` que tienes actualmente, Git no recupera (fetch) ningún dato; simplemente prepara una rama remota llamada `teamone/master` para apuntar a la confirmación (commit) que `teamone` tiene en su rama `master`.

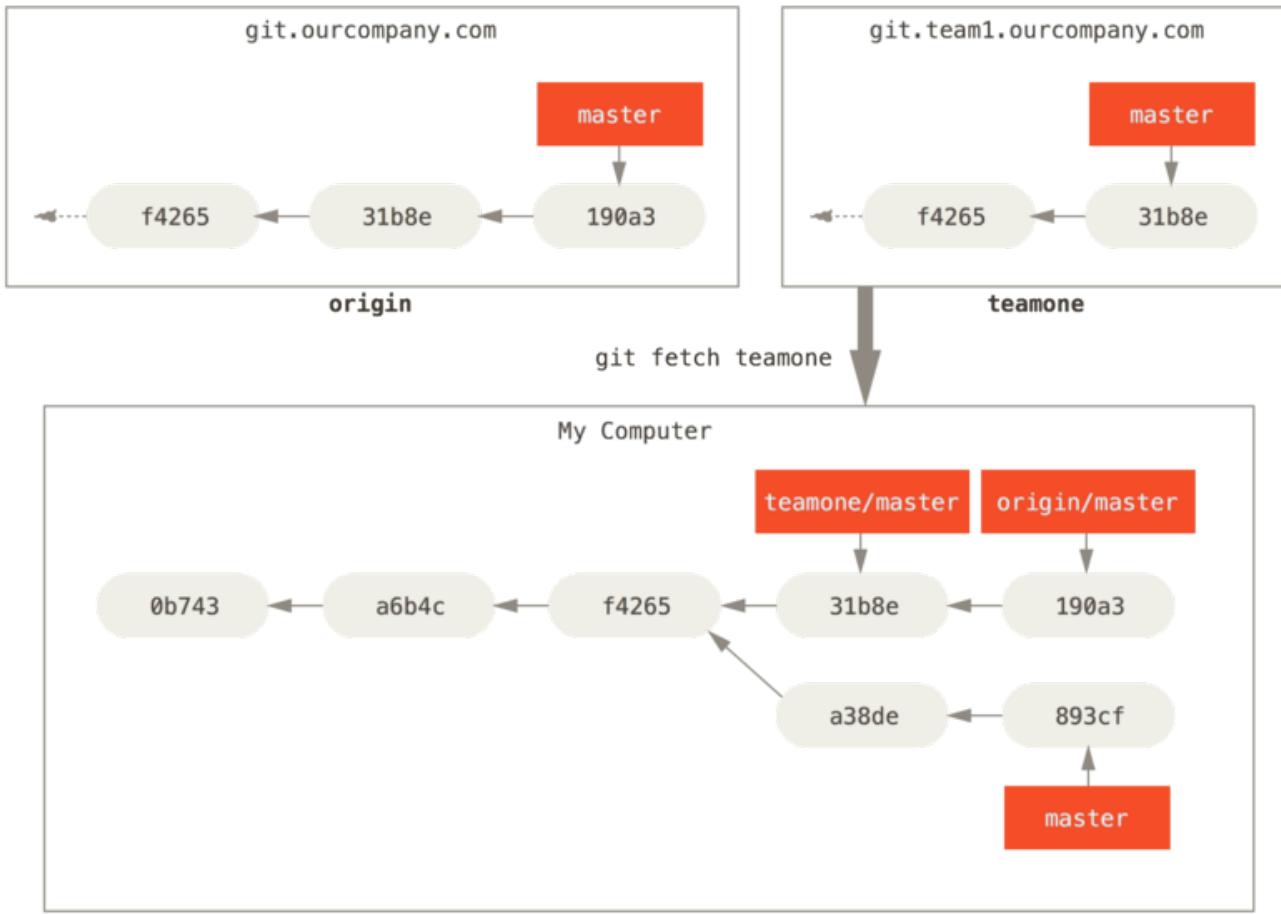


Figure 34. Seguimiento de la rama remota a través de `teamone/master`

## Publicar

Cuando quieres compartir una rama con el resto del mundo, debes llevarla (`push`) a un remoto donde tengas permisos de escritura. Tus ramas locales no se sincronizan automáticamente con los remotos en los que escribes, sino que tienes que enviar (`push`) expresamente las ramas que deseas compartir. De esta forma, puedes usar ramas privadas para el trabajo que no deseas compartir, llevando a un remoto tan solo aquellas partes que deseas aportar a los demás.

Si tienes una rama llamada `serverfix`, con la que vas a trabajar en colaboración; puedes llevarla al remoto de la misma forma que llevaste tu primera rama. Con el comando `git push (remoto) (rama)`:

```
$ git push origin serverfix
Counting objects: 24, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (15/15), done.
Writing objects: 100% (24/24), 1.91 KiB | 0 bytes/s, done.
Total 24 (delta 2), reused 0 (delta 0)
To https://github.com/schacon/simplegit
 * [new branch]      serverfix -> serverfix
```

Esto es un atajo. Git expande automáticamente el nombre de rama `serverfix` a `refs/heads/serverfix:refs/heads/serverfix`, que significa: “coge mi rama local `serverfix` y actualiza con ella la rama `serverfix` del remoto”. Volveremos más tarde sobre el tema de `refs/heads/`,

viéndolo en detalle en [Los entresijos internos de Git](#); por ahora, puedes ignorarlo. También puedes hacer `git push origin serverfix:serverfix`, que hace lo mismo; es decir: “coge mi `serverfix` y hazlo el `serverfix` remoto”. Puedes utilizar este último formato para llevar una rama local a una rama remota con un nombre distinto. Si no quieres que se llame `serverfix` en el remoto, puedes lanzar, por ejemplo, `git push origin serverfix:awesomebranch`; para llevar tu rama `serverfix` local a la rama `awesomebranch` en el proyecto remoto.

#### *No escribas tu contraseña todo el tiempo*

Si utilizas una dirección URL con HTTPS para enviar datos, el servidor Git te preguntará tu usuario y contraseña para autenticarte. Por defecto, te pedirá esta información a través del terminal, para determinar si estás autorizado a enviar datos.

#### NOTE

Si no quieres escribir tu contraseña cada vez que haces un envío, puedes establecer un “cache de credenciales”. La manera más sencilla de hacerlo es estableciéndolo en memoria por unos minutos, lo que puedes lograr fácilmente al ejecutar `git config --global credential.helper cache`

Para más información sobre las distintas opciones de cache de credenciales, véase [Almacenamiento de credenciales](#).

La próxima vez que tus colaboradores recuperen desde el servidor, obtendrán bajo la rama remota `origin/serverfix` una referencia a donde esté la versión de `serverfix` en el servidor:

```
$ git fetch origin
remote: Counting objects: 7, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 3 (delta 0)
Unpacking objects: 100% (3/3), done.
From https://github.com/schacon/simplegit
 * [new branch]      serverfix    -> origin/serverfix
```

Es importante destacar que cuando recuperas (fetch) nuevas ramas remotas, no obtienes automáticamente una copia local editable de las mismas. En otras palabras, en este caso, no tienes una nueva rama `serverfix`. Sino que únicamente tienes un puntero no editable a `origin/serverfix`.

Para integrar (merge) esto en tu rama de trabajo actual, puedes usar el comando `git merge origin/serverfix`. Y si quieres tener tu propia rama `serverfix` para trabajar, puedes crearla directamente basandote en la rama remota:

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

Esto sí te da una rama local donde puedes trabajar, que comienza donde `origin/serverfix` estaba en ese momento.

## Hacer Seguimiento a las Ramas

Al activar (checkout) una rama local a partir de una rama remota, se crea automáticamente lo que podríamos denominar una “rama de seguimiento” (tracking branch). Las ramas de seguimiento son ramas locales que tienen una relación directa con alguna rama remota. Si estás en una rama de seguimiento y tecleas el comando `git pull`, Git sabe de cuál servidor recuperar (fetch) y fusionar (merge) datos.

Cuando clonas un repositorio, este suele crear automáticamente una rama `master` que hace seguimiento de `origin/master`. Sin embargo, puedes preparar otras ramas de seguimiento si deseas tener unas que sigan ramas de otros remotos o no seguir la rama `master`. El ejemplo más simple es el que acabas de ver al lanzar el comando `git checkout -b [rama] [nombreremoto]/[rama]`. Esta operación es tan común que git ofrece el parámetro `--track`:

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

Para preparar una rama local con un nombre distinto a la del remoto, puedes utilizar la primera versión con un nombre de rama local diferente:

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

Así, tu rama local `sf` traerá (pull) información automáticamente desde `origin/serverfix`.

Si ya tienes una rama local y quieres asignarla a una rama remota que acabas de traerte, o quieres cambiar la rama a la que le haces seguimiento, puedes usar en cualquier momento las opciones `-u` o `--set-upstream-to` del comando `git branch`.

```
$ git branch -u origin/serverfix
Branch serverfix set up to track remote branch serverfix from origin.
```

### *Atajo al upstream*

**NOTE** Cuando tienes asignada una rama de seguimiento, puedes hacer referencia a ella mediante `@{upstream}` o mediante el atajo `@{u}`. De esta manera, si estás en la rama `master` y esta sigue a la rama `origin/master`, puedes hacer algo como `git merge @{u}` en vez de `git merge origin/master`.

Si quieres ver las ramas de seguimiento que tienes asignado, puedes usar la opción `-vv` con `git branch`. Esto listará tus ramas locales con más información, incluyendo a qué sigue cada rama y si tu rama local está por delante, por detrás o ambas.

```
$ git branch -vv
iss53    7e424c3 [origin/iss53: ahead 2] forgot the brackets
master    1ae2a45 [origin/master] deploying index fix
* serverfix f8674d9 [teamone/server-fix-good: ahead 3, behind 1] this should do it
  testing   5ea463a trying something new
```

Aquí podemos ver que nuestra rama `iss53` sigue `origin/iss53` y está “ahead” (delante) por dos, es decir, que tenemos dos confirmaciones locales que no han sido enviadas al servidor. También podemos ver que nuestra rama `master` sigue a `origin/master` y está actualizada. Luego podemos ver que nuestra rama `serverfix` sigue la rama `server-fix-good` de nuestro servidor `teamone` y que está tres cambios por delante (ahead) y uno por detrás (behind), lo que significa que existe una confirmación en el servidor que no hemos fusionado y que tenemos tres confirmaciones locales que no hemos enviado. Por último, podemos ver que nuestra rama `testing` no sigue a ninguna rama remota.

Es importante destacar que estos números se refieren a la última vez que trajiste (fetch) datos de cada servidor. Este comando no se comunica con los servidores, solo te indica lo que sabe de ellos localmente. Si quieres tener los cambios por delante y por detrás actualizados, debes traértelos (fetch) de cada servidor antes de ejecutar el comando. Puedes hacerlo de esta manera: `$ git fetch --all; git branch -vv`

## Traer y Fusionar

A pesar de que el comando `git fetch` trae todos los cambios del servidor que no tienes, este no modifica tu directorio de trabajo. Simplemente obtendrá los datos y dejará que tú mismo los fusiones. Sin embargo, existe un comando llamado `git pull`, el cuál básicamente hace `git fetch` seguido por `git merge` en la mayoría de los casos. Si tienes una rama de seguimiento configurada como vimos en la última sección, bien sea asignándola explícitamente o creándola mediante los comandos `clone` o `checkout`, `git pull` identificará a qué servidor y rama remota sigue tu rama actual, traerá los datos de dicho servidor e intentará fusionar dicha rama remota.

Normalmente es mejor usar los comandos `fetch` y `merge` de manera explícita pues la magia de `git pull` puede resultar confusa.

## Eliminar Ramas Remotas

Imagina que ya has terminado con una rama remota, es decir, tanto tú como tus colaboradores habéis completado una determinada funcionalidad y la habéis incorporado (merge) a la rama `master` en el remoto (o donde quiera que tengáis la rama de código estable). Puedes borrar la rama remota utilizando la opción `--delete` de `git push`. Por ejemplo, si quieres borrar la rama `serverfix` del servidor, puedes utilizar:

```
$ git push origin --delete serverfix
To https://github.com/schacon/simplegit
 - [deleted]           serverfix
```

Básicamente lo que hace es eliminar el apuntador del servidor. El servidor Git suele mantener los

datos por un tiempo hasta que el recolector de basura se ejecute, de manera que si la has borrado accidentalmente, suele ser fácil recuperarla.

## ~~Reorganizar el Trabajo Realizado~~

En Git tenemos dos formas de integrar cambios de una rama en otra: la fusión (merge) y la reorganización (rebase). En esta sección vas a aprender en qué consiste la reorganización, cómo utilizarla, por qué es una herramienta sorprendente y en qué casos no es conveniente utilizarla.

### Reorganización Básica

Volviendo al ejemplo anterior, en la sección sobre fusiones [Procedimientos Básicos de Fusión](#) puedes ver que has separado tu trabajo y realizado confirmaciones (commit) en dos ramas diferentes.

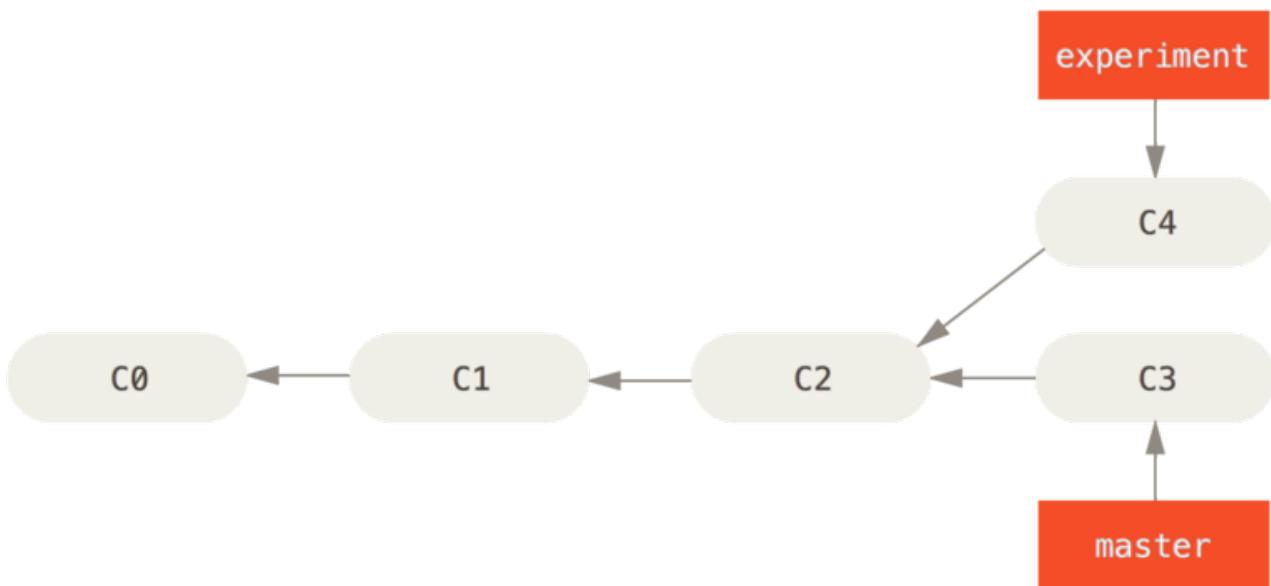


Figure 35. El registro de confirmaciones inicial

La manera más sencilla de integrar ramas, tal y como hemos visto, es el comando `git merge`. Realiza una fusión a tres bandas entre las dos últimas instantáneas de cada rama (C3 y C4) y el ancestro común a ambas (C2); creando una nueva instantánea (snapshot) y la correspondiente confirmación (commit).

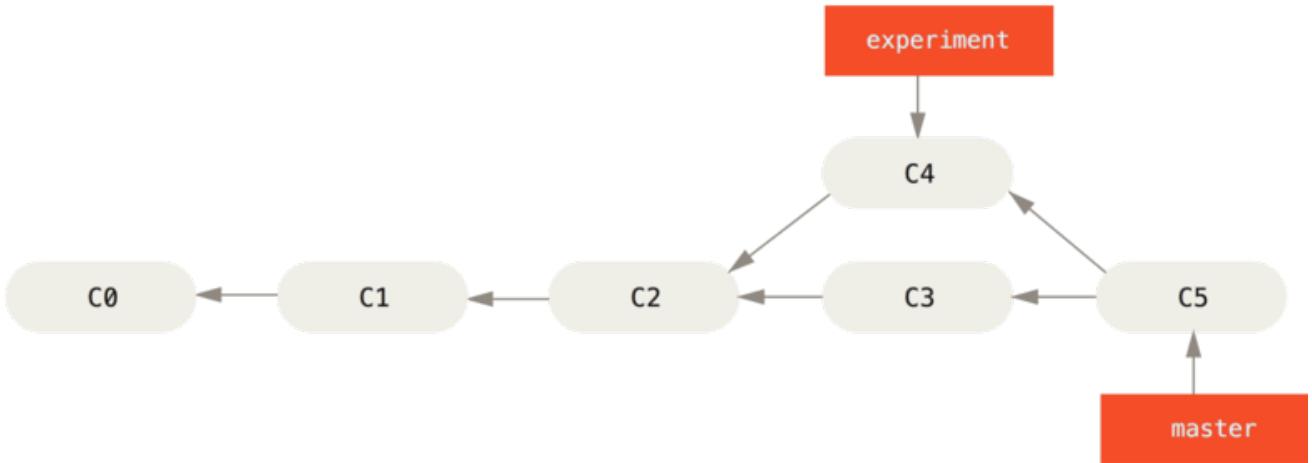


Figure 36. Fusionar una rama para integrar el registro de trabajos divergentes

Sin embargo, también hay otra forma de hacerlo: puedes coger los cambios introducidos en C3 y reaplicarlos encima de C4. Esto es lo que en Git llamamos *reorganizar* (*rebasing*, en inglés). Con el comando `git rebase`, puedes coger todos los cambios confirmados en una rama, y reaplicarlos sobre otra.

Por ejemplo, puedes lanzar los comandos:

```

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

```

Haciendo que Git vaya al ancestro común de ambas ramas (donde estás actualmente y de donde quieras reorganizar), saque las diferencias introducidas por cada confirmación en la rama donde estás, guarde esas diferencias en archivos temporales, reinicie (reset) la rama actual hasta llevarla a la misma confirmación en la rama de donde quieras reorganizar, y, finalmente, vuelva a aplicar ordenadamente los cambios.

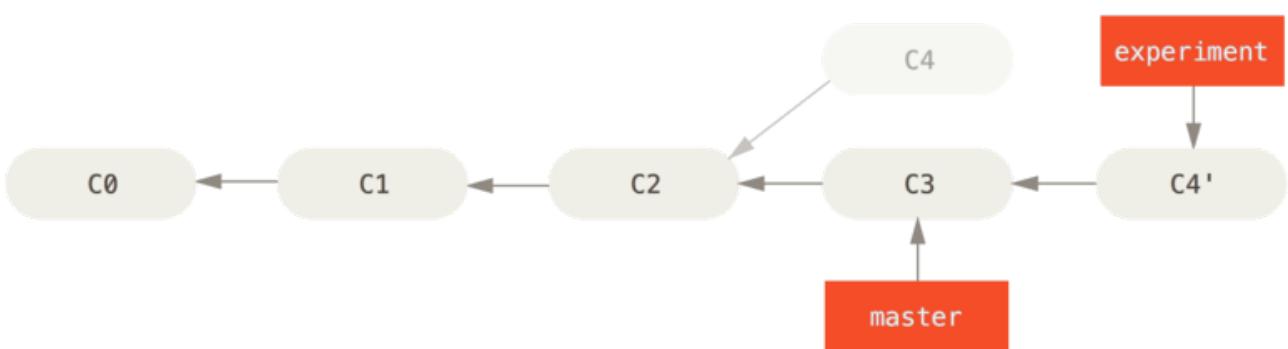


Figure 37. Reorganizando sobre C3 los cambios introducidos en C4

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

```
$ git checkout master  
$ git merge experiment
```

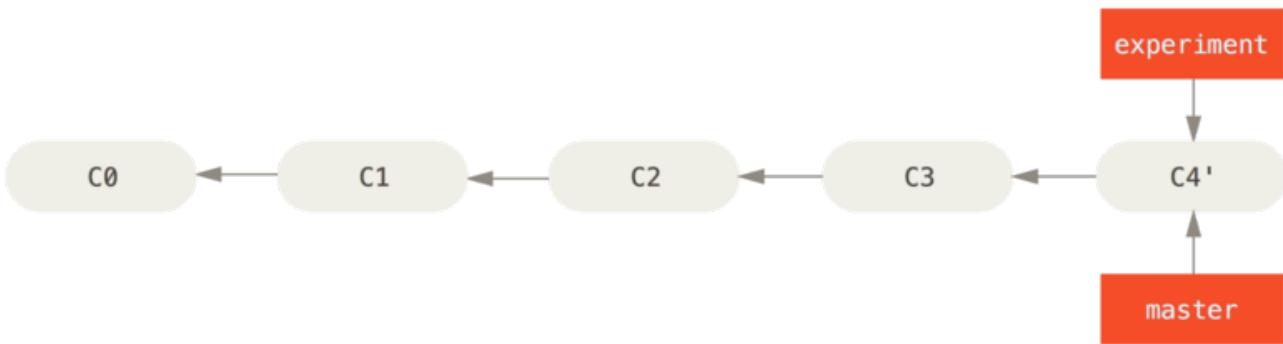


Figure 38. Avance rápido de la rama `master`

Así, la instantánea apuntada por `C4'` es exactamente la misma apuntada por `C5` en el ejemplo de la fusión. No hay ninguna diferencia en el resultado final de la integración, pero el haberla hecho reorganizando nos deja un historial más claro. Si examinas el historial de una rama reorganizada, este aparece siempre como un historial lineal: como si todo el trabajo se hubiera realizado en series, aunque realmente se haya hecho en paralelo.

Habitualmente, optarás por esta vía cuando quieras estar seguro de que tus confirmaciones de cambio (commits) se pueden aplicar limpiamente sobre una rama remota; posiblemente, en un proyecto donde estés intentando colaborar, pero lleves tú el mantenimiento. En casos como esos, puedes trabajar sobre una rama y luego reorganizar lo realizado en la rama `origin/master` cuando lo tengas todo listo para enviarlo al proyecto principal. De esta forma, la persona que mantiene el proyecto no necesitará hacer ninguna integración con tu trabajo; le bastará con un avance rápido o una incorporación limpia.

Cabe destacar que la instantánea (snapshot) apuntada por la confirmación (commit) final, tanto si es producto de una reorganización (rebase) como si lo es de una fusión (merge), es exactamente la misma instantánea; lo único diferente es el historial. La reorganización vuelve a aplicar cambios de una rama de trabajo sobre otra rama, en el mismo orden en que fueron introducidos en la primera, mientras que la fusión combina entre sí los dos puntos finales de ambas ramas.

## Algunas Reorganizaciones Interesantes

También puedes aplicar una reorganización (rebase) sobre otra cosa además de sobre la rama de reorganización. Por ejemplo, considera un historial como el de [Un historial con una rama puntual sobre otra rama puntual](#). Has ramificado a una rama puntual (`server`) para añadir algunas funcionalidades al proyecto, y luego has confirmado los cambios. Despues, vuelves a la rama original para hacer algunos cambios en la parte cliente (rama `client`), y confirmas también esos cambios. Por último, vuelves sobre la rama `server` y haces algunos cambios más.

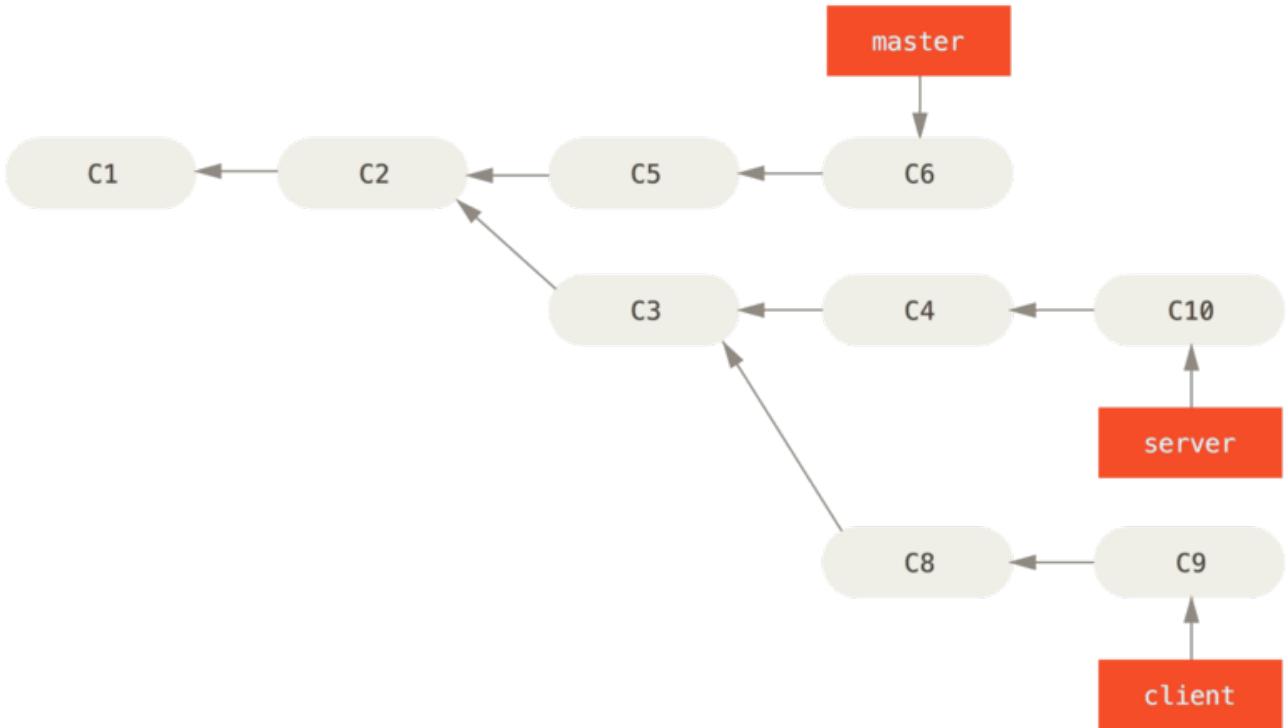


Figure 39. Un historial con una rama puntual sobre otra rama puntual

Imagina que decides incorporar tus cambios del lado cliente sobre el proyecto principal para hacer un lanzamiento de versión; pero no quieres lanzar aún los cambios del lado servidor porque no están aún suficientemente probados. Puedes coger los cambios del cliente que no están en server (C8 y C9) y reaplicarlos sobre tu rama principal usando la opción `--onto` del comando `git rebase`:

```
$ git rebase --onto master server client
```

Esto viene a decir: “Activa la rama `client`, averigua los cambios desde el ancestro común entre las ramas `client` y `server`, y aplícalos en la rama `master`”. Puede parecer un poco complicado, pero los resultados son realmente interesantes.

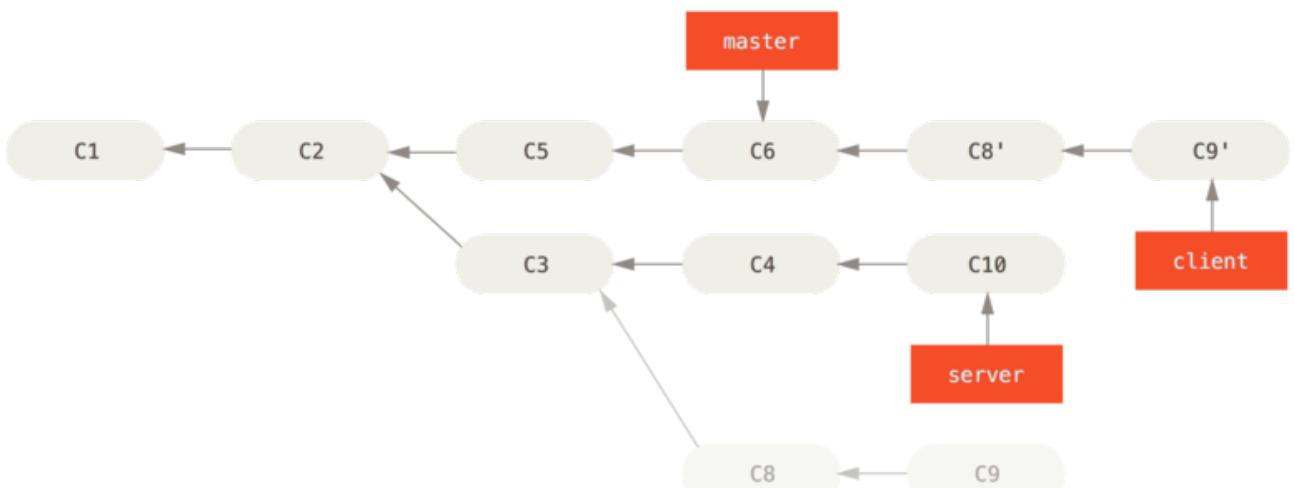


Figure 40. Reorganizando una rama puntual fuera de otra rama puntual

Y, tras esto, ya puedes avanzar la rama principal (ver [Avance rápido de tu rama master](#), para incluir

los cambios de la rama `client`):

```
$ git checkout master  
$ git merge client
```

Figure 41. Avance rápido de tu rama `master`, para incluir los cambios de la rama `client`

Ahora supongamos que decides traerlos (pull) también sobre tu rama `server`. Puedes reorganizar (rebase) la rama `server` sobre la rama `master` sin necesidad siquiera de comprobarlo previamente, usando el comando `git rebase [rama-base] [rama-puntual]`, el cual activa la rama puntual (`server` en este caso) y la aplica sobre la rama base (`master` en este caso):

```
$ git rebase master server
```

Esto vuelca el trabajo de `server` sobre el de `master`, tal y como se muestra en [Reorganizando la rama `server` sobre la rama `master`](#).

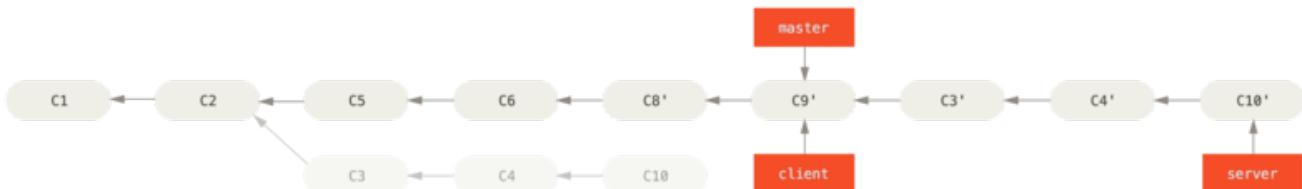


Figure 42. Reorganizando la rama `server` sobre la rama `master`

Después, puedes avanzar rápidamente la rama base (`master`):

```
$ git checkout master  
$ git merge server
```

Y por último puedes eliminar las ramas `client` y `server` porque ya todo su contenido ha sido integrado y no las vas a necesitar más, dejando tu registro tras todo este proceso tal y como se muestra en [Historial final de confirmaciones de cambio](#):

```
$ git branch -d client  
$ git branch -d server
```



Figure 43. Historial final de confirmaciones de cambio

## Los Peligros de Reorganizar

Ahh..., pero la dicha de la reorganización no la alcanzamos sin sus contrapartidas, las cuales

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, la gente te odiará y serás despreciado por tus familiares y amigos.

Cuando reorganizas algo, estás abandonando las confirmaciones de cambio ya creadas y estás creando unas nuevas; que son similares, pero diferentes. Si envías (push) confirmaciones (commits) a alguna parte, y otros las recogen (pull) de allí; y después vas tú y las reescribes con `git rebase` y las vuelves a enviar (push); tus colaboradores tendrán que refusionar (re-merge) su trabajo y todo se volverá tremadamente complicado cuando intentes recoger (pull) su trabajo de vuelta sobre el tuyo.

Veamos con un ejemplo como reorganizar trabajo que has hecho público puede causar problemas. Imagínate que haces un clon desde un servidor central, y luego trabajas sobre él. Tu historial de cambios puede ser algo como esto:

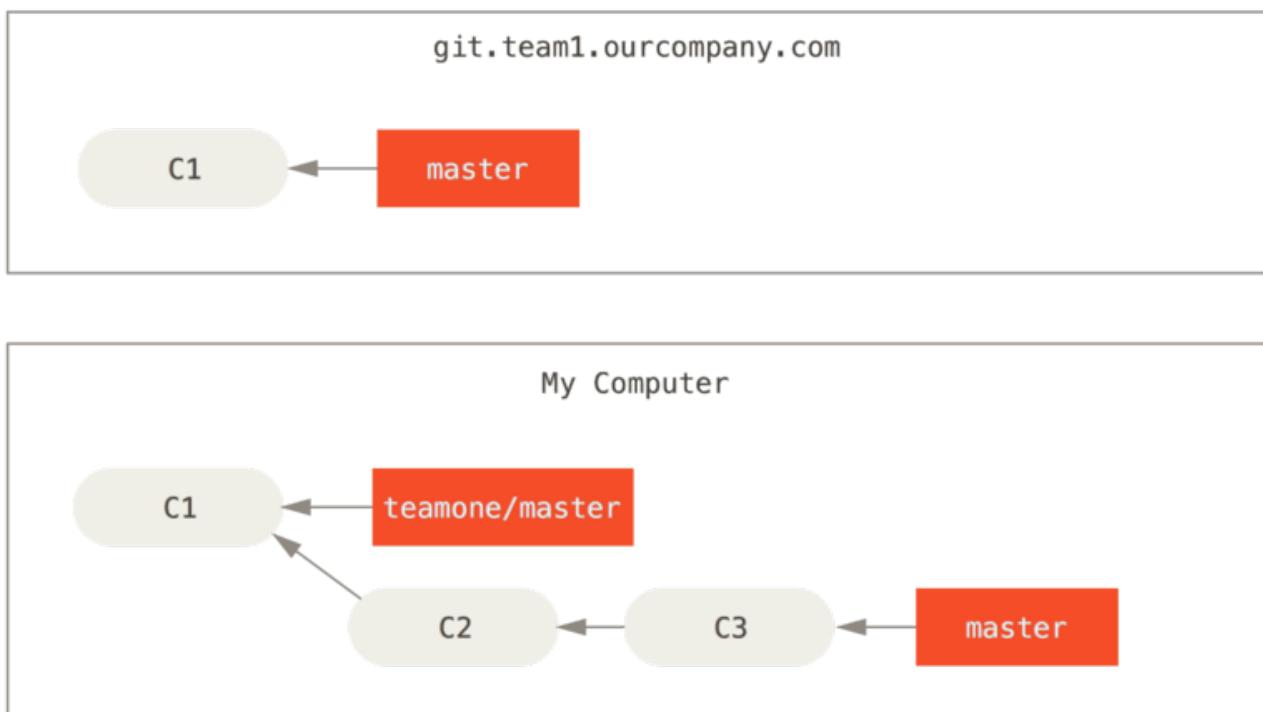
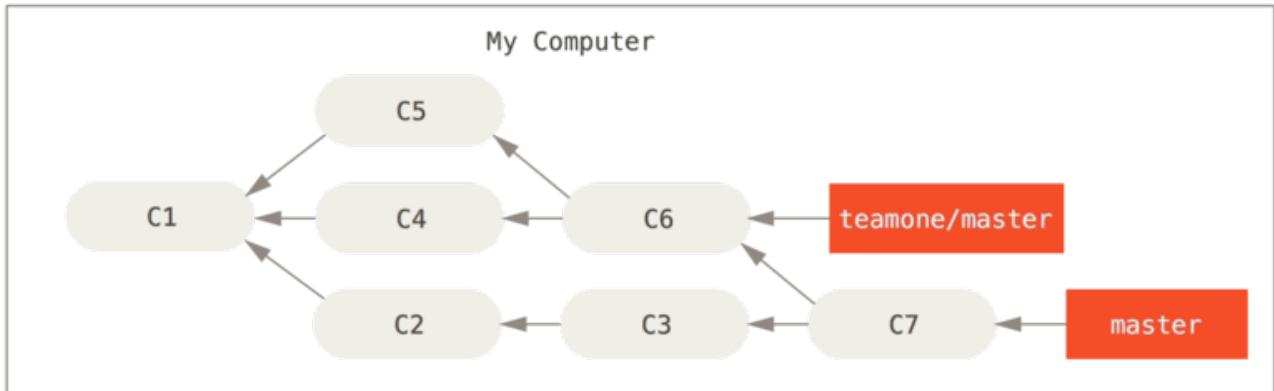
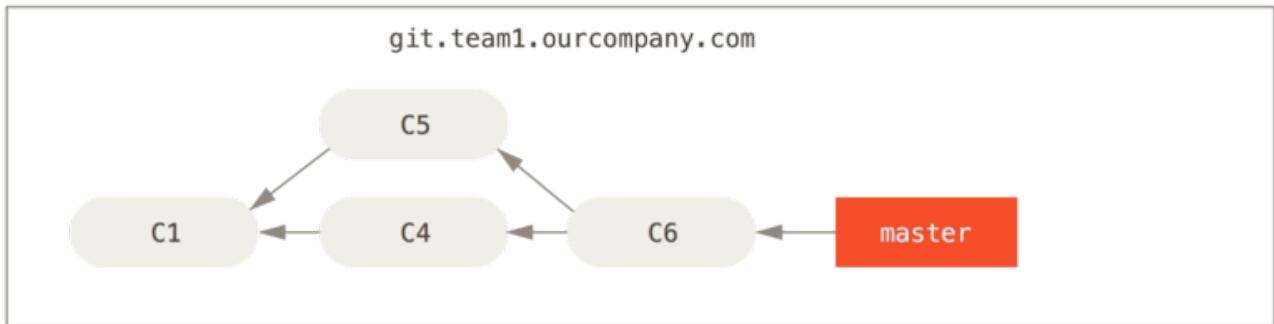


Figure 44. Clonar un repositorio y trabajar sobre él

Ahora, otra persona trabaja también sobre ello, realiza una fusión (merge) y lleva (push) su trabajo al servidor central. Tú te traes (fetch) sus trabajos y los fusionas (merge) sobre una nueva rama en tu trabajo, con lo que tu historial quedaría parecido a esto:



*Figure 45. Traer (fetch) algunas confirmaciones de cambio (commits) y fusionarlas (merge) sobre tu trabajo*

A continuación, la persona que había llevado cambios al servidor central decide retroceder y reorganizar su trabajo; haciendo un `git push --force` para sobrescribir el registro en el servidor. Tu te traes (fetch) esos nuevos cambios desde el servidor.

*Figure 46. Alguien envio (push) confirmaciones (commits) reorganizadas, abandonando las confirmaciones en las que tu habías basado tu trabajo*

Ahora los dos están en un aprieto. Si haces `git pull` crearás una fusión confirmada, la cual incluirá ambas líneas del historial, y tu repositorio lucirá así:

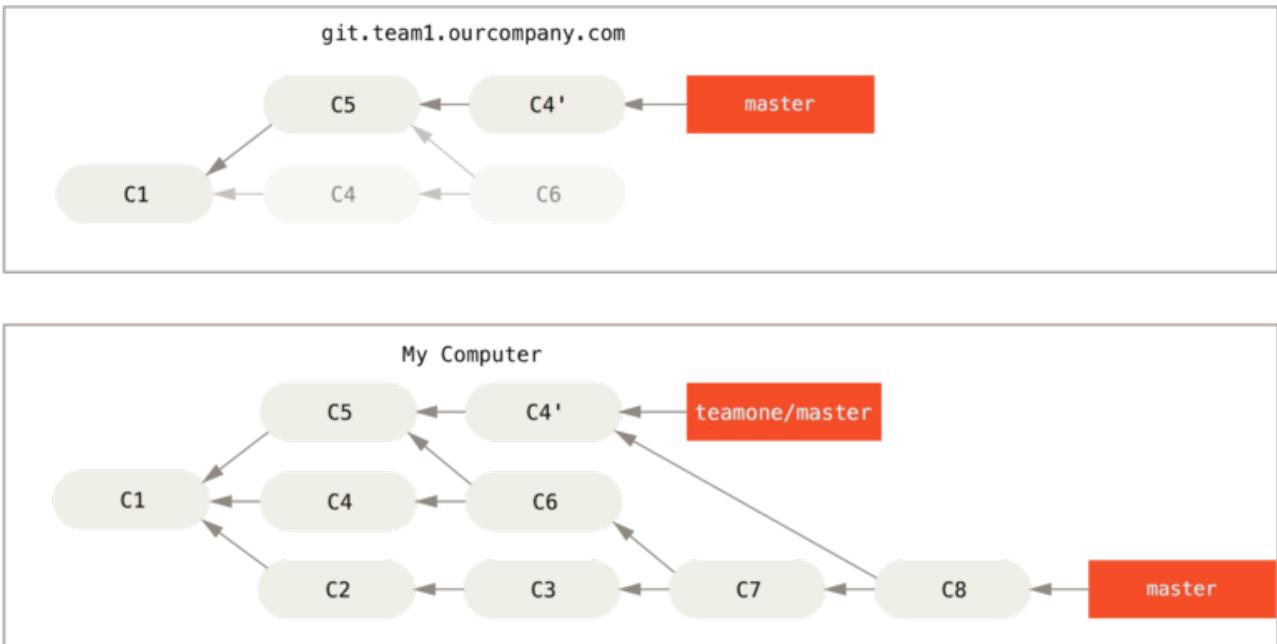


Figure 47. Vuelves a fusionar el mismo trabajo en una nueva fusión confirmada

Si ejecutas `git log` sobre un historial así, verás dos confirmaciones hechas por el mismo autor y con la misma fecha y mensaje, lo cual será confuso. Es más, si luego tu envías (push) ese registro de vuelta al servidor, vas a introducir todas esas confirmaciones reorganizadas en el servidor central. Lo que puede confundir aún más a la gente. Era más seguro asumir que el otro desarrollador no quería que `C4` y `C6` estuviesen en el historial; por ello había reorganizado su trabajo de esa manera.

## Reorganizar una Reorganización

Si te encuentras en una situación como esta, Git tiene algunos trucos que pueden ayudarte. Si alguien de tu equipo sobreescribe cambios en los que se basaba tu trabajo, tu reto es descubrir qué han sobreescrito y qué te pertenece.

Además de la suma de control SHA-1, Git calcula una suma de control basada en el parche que introduce una confirmación. A esta se le conoce como “patch-id”.

Si te traes el trabajo que ha sido sobreescrito y lo reorganizas sobre las nuevas confirmaciones de tu compañero, es posible que Git pueda identificar qué parte correspondía específicamente a tu trabajo y aplicarla de vuelta en la rama nueva.

Por ejemplo, en el caso anterior, si en vez de hacer una fusión cuando estábamos en [Alguien envio \(push\) confirmaciones \(commits\) reorganizadas, abandonando las confirmaciones en las que tu habías basado tu trabajo](#) ejecutamos `git rebase teamone/master`, Git hará lo siguiente:

- Determinar el trabajo que es específico de nuestra rama (C2, C3, C4, C6, C7)
- Determinar cuáles no son fusiones confirmadas (C2, C3, C4)
- Determinar cuáles no han sido sobreescritas en la rama destino (solo C2 y C3, pues C4 corresponde al mismo parche que C4')
- Aplicar dichas confirmaciones encima de `teamone/master`

Así que en vez del resultado que vimos en [Vuelves a fusionar el mismo trabajo en una nueva fusión confirmada](#), terminaremos con algo más parecido a [Reorganizar encima de un trabajo sobreescrito reorganizado..](#)

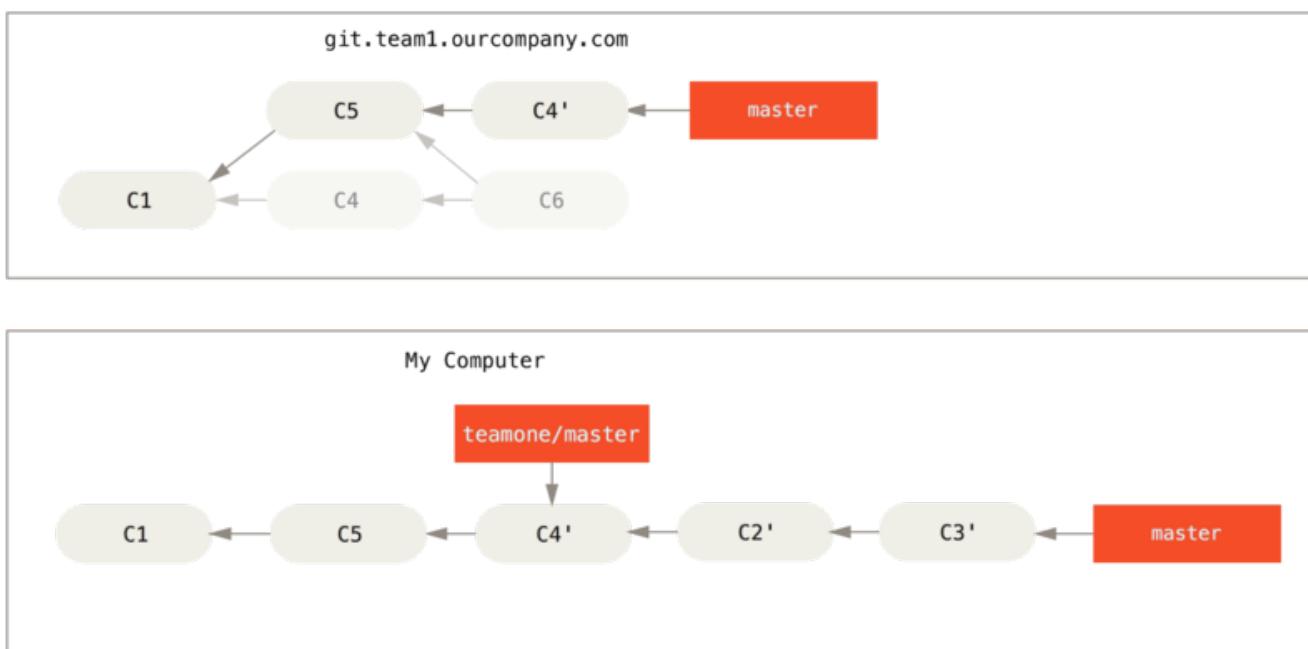


Figure 48. Reorganizar encima de un trabajo sobreescrito reorganizado.

Esto solo funciona si C4 y el C4' de tu compañero son parches muy similares. De lo contrario, la reorganización no será capaz de identificar que se trata de un duplicado y agregará otro parche similar a C4 (lo cual probablemente no podrá aplicarse limpiamente, pues los cambios ya estarán allí en algún lugar).

También puedes simplificar el proceso si ejecutas `git pull --rebase` en vez del tradicional `git pull`. O, en este caso, puedes hacerlo manualmente con un `git fetch` primero, seguido de un `git rebase teamone/master`.

Si sueles utilizar `git pull` y quieres que la opción `--rebase` esté activada por defecto, puedes asignar el valor de configuración `pull.rebase` haciendo algo como esto `git config --global pull.rebase true`.

Si consideras la reorganización como una manera de limpiar tu trabajo y tus confirmaciones antes de enviarlas (push), y si solo reorganizas confirmaciones (commits) que nunca han estado disponibles públicamente, no tendrás problemas. Si reorganizas (rebase) confirmaciones (commits) que ya estaban disponibles públicamente y la gente había basado su trabajo en ellas, entonces prepárate para tener problemas, frustrar a tu equipo y ser despreciado por tus compañeros.

Si tu compañero o tú ven que aun así es necesario hacerlo en algún momento, asegúrense que todos sepan que deben ejecutar `git pull --rebase` para intentar aliviar en lo posible la frustración.

## Reorganizar vs. Fusionar

Ahora que has visto en acción la reorganización y la fusión, te preguntarás cuál es mejor. Antes de responder, repasemos un poco qué representa el historial.

Para algunos, 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.

Ahora, sobre qué es mejor si fusionar o reorganizar: verás que la respuesta no es tan sencilla. Git es una herramienta poderosa que te permite hacer muchas cosas con tu historial, y cada equipo y cada proyecto es diferente. Ahora que conoces cómo trabajan ambas herramientas, será cosa tuya decidir cuál de las dos es mejor para tu situación en particular.

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

## Recapitulación

Hemos visto los procedimientos básicos de ramificación (branching) y fusión (merging) en Git. A estas alturas, te sentirás cómodo creando nuevas ramas (branch), saltando (checkout) entre ramas para trabajar y fusionando (merge) ramas entre ellas. También conocerás cómo compartir tus ramas enviándolas (push) a un servidor compartido, cómo trabajar colaborativamente en ramas compartidas, y cómo reorganizar (rebase) tus ramas antes de compartirlas. A continuación, hablaremos sobre lo que necesitas para tener tu propio servidor de hospedaje Git.