Módulo 3. Fusión de ramas y resolución de conflictos

CÓRDOBA CREA FUTURO

- Introducción
- 1. Inicio Git
- 2. Ramificaciones Git
- 3. Cambio de archivos al saltar entre ramas
- 4. Ramificaciones en Git: ramificar y fusionar
- Referencias

Introducción

La fusión de ramas en Git (*git merge*) integra el historial y los cambios de una rama secundaria en otra —por lo general, *main* o *master*— con el fin de unir líneas de desarrollo independientes. Los conflictos de fusión se producen cuando existen cambios contrapuestos en la misma línea de un archivo, lo que impide la combinación automática y requiere intervención manual para decidir qué modificaciones conservar.

La fusión de ramas tiene como propósito combinar el trabajo desarrollado por separado en una rama funcional con la rama principal del proyecto. Para ello, suele utilizarse el comando `git merge <rama-a-fusionar>` mientras se está ubicado en la rama de destino. Este proceso puede realizarse mediante un *fast-forward* —cuando no hay divergencia entre ramas— o mediante un *true merge*, que genera un nuevo commit de fusión.

Los conflictos suelen aparecer cuando se modifican de forma diferente las mismas líneas de un archivo en dos ramas o cuando una elimina un archivo que la otra modifica. En estos casos, Git detiene la fusión y marca los archivos en conflicto. Es necesario editar manualmente los archivos, seleccionar las líneas correctas, eliminar los marcadores de conflicto (`<<<<<<<`, `=======`, `>>>>>>>`), agregar los cambios con `git add` y completar el proceso con `git commit`. La resolución puede realizarse desde la línea de comandos, en editores de código como VS Code o mediante herramientas gráficas de Git.

Para gestionar una fusión de manera ordenada, primero se actualiza la rama base con `git checkout main` y `git pull`. Luego se ejecuta `git merge <rama-funcional>`. Si surgen conflictos, se abren los archivos afectados, se corrige el código y se guardan los cambios. Finalmente, se agregan los archivos modificados y se confirma la fusión con `git commit`.

Antes de continuar, se presenta una breve referencia histórica que servirá como punto de partida para la explicación formal del tema. El desarrollo se inspira en la bibliografía que se consigna al final.

CONTINUAR

1. Inicio Git

Como muchas de las grandes innovaciones, Git surgió en un contexto de cambios profundos y controversias. El *kernel* de Linux es un proyecto de *software* de código abierto de gran alcance. Durante la mayor parte del mantenimiento del sistema operativo Linux, entre 1991 y 2002, las modificaciones se gestionaban mediante parches y archivos. En 2002, el proyecto comenzó a utilizar un sistema de control de versiones distribuido (DVCS) propietario llamado BitKeeper.

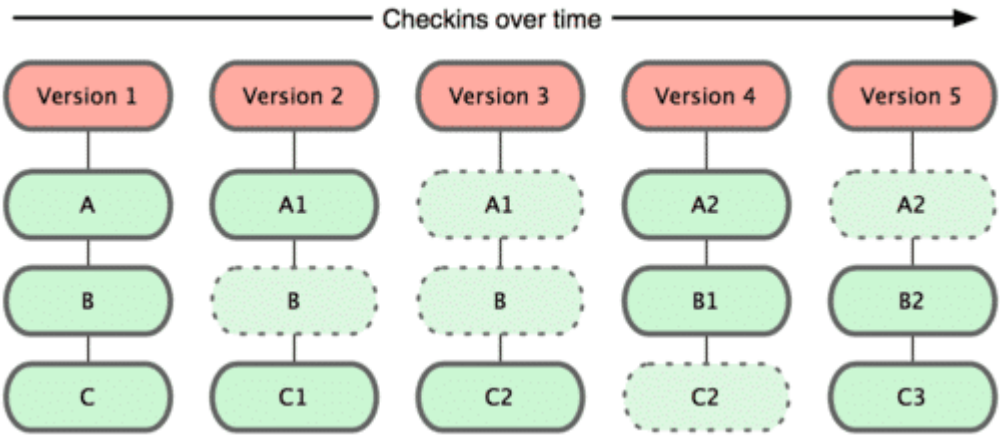
En 2005, la relación entre la comunidad que desarrollaba el kernel de Linux y la empresa responsable de BitKeeper se deterioró, por lo que la herramienta dejó de ofrecerse de forma gratuita. Esta situación impulsó a la comunidad de desarrollo —y, en particular, a Linus Torvalds, creador de Linux— a diseñar su propia herramienta, tomando como referencia la experiencia adquirida con BitKeeper.

El nuevo sistema se propuso cumplir con los siguientes objetivos:

- alta velocidad;
- diseño sencillo;
- amplio soporte para el desarrollo no lineal, con miles de ramas paralelas;
- funcionamiento completamente distribuido;
- capacidad para gestionar proyectos de gran tamaño, como el kernel de Linux, de manera eficiente en términos de velocidad y almacenamiento de datos.

Desde su creación en 2005, Git ha evolucionado y madurado hasta convertirse en una herramienta fácil de usar que conserva sus características originales. Es rápido, eficiente en proyectos de gran escala y cuenta con un sólido sistema de ramificación (*branching*) que facilita el desarrollo no lineal.

Figura 1. Versiones distintas de actualización.



Fuente: GitHub, s.f., https://goo.su/nqMV1

SISTEMA DE CONTROL DE VERSIONES | **SISTEMAS DE CONTROL DE VERSIONES CENTRALIZADOS** | **SISTEMAS DE CONTROL DE VERSIONES DISTRIBUIDOS**

Un VCS (*version control system* o sistema de control de versiones) es una herramienta de software diseñada para rastrear, gestionar y registrar los cambios en archivos —especialmente en código fuente— a lo largo del tiempo. Permite trabajar en equipo, recuperar versiones anteriores, comparar modificaciones y colaborar sin sobrescribir el trabajo de otras personas.

En el caso de Git, se trata de un sistema distribuido (DVCS), lo que significa que cada desarrollador cuenta con una copia completa del repositorio, incluido su historial, en el entorno local.

Entre sus principales características se encuentran las siguientes:

- Registro detallado del historial, que permite conocer quién realizó un cambio, cuándo y en qué archivo.
- Funcionamiento distribuido, ya que cada integrante dispone de una copia íntegra del repositorio.

- Gestión de ramas (*branches*) y fusiones (*merge*), lo que facilita el trabajo paralelo en distintas funcionalidades sin afectar la rama principal (*main* o *master*).
- Posibilidad de revertir cambios y recuperar estados anteriores ante errores.
- Integración de modificaciones realizadas por diferentes personas en un mismo proyecto.

El control de versiones, en términos generales, es un sistema que registra los cambios en uno o varios archivos con el propósito de poder recuperar versiones específicas más adelante. Aunque suele emplearse para el código fuente, puede utilizarse con casi cualquier tipo de archivo en una computadora.

Su importancia radica en que permite revertir archivos seleccionados o incluso todo un proyecto a un estado previo, comparar cambios a lo largo del tiempo, identificar quién realizó una modificación determinada y en qué momento, y detectar el origen de un problema. Asimismo, si se pierde información o se produce un error, es posible recuperar versiones anteriores con facilidad y sin requerir grandes recursos.

Por otro lado, los sistemas de control de versiones locales surgieron como una mejora frente a la práctica de crear múltiples copias manuales de archivos o carpetas —por ejemplo, en trabajos académicos como tesis o en distintas versiones de un software—. Si bien este método resulta sencillo, es propenso a errores: puede olvidarse en qué directorio se está trabajando, sobrescribir un archivo equivocado o duplicar documentos innecesarios.

Figura 2. Ejemplo de organización de archivos con múltiples versiones locales

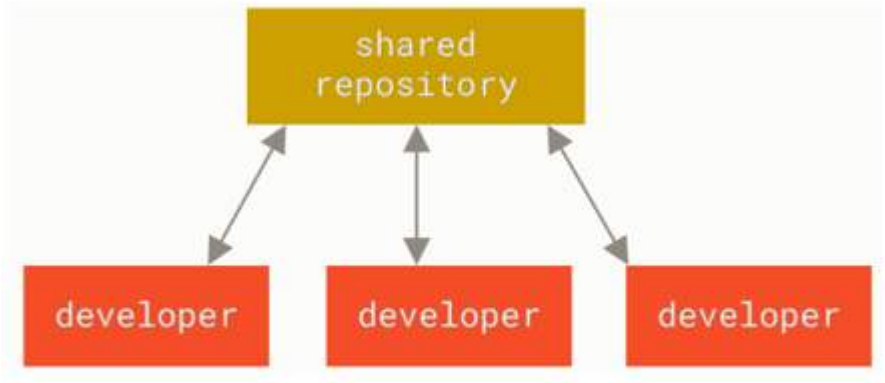
Fuente: elaboración propia

Los sistemas de control de versiones centralizados se caracterizan por contar con un único servidor que almacena todos los archivos versionados, mientras que diversos clientes obtienen la información desde ese servidor central.

Una de sus ventajas es que permite que todas las personas involucradas conozcan, en cierta medida, el trabajo que realizan las demás dentro del proyecto. Además, los administradores pueden ejercer un control preciso sobre los permisos de acceso y las acciones que cada integrante puede llevar a cabo. La gestión de un sistema centralizado también suele resultar más sencilla que la administración de múltiples bases de datos locales en cada equipo.

No obstante, este modelo presenta desventajas. La principal es que el servidor central constituye un punto único de fallo. Si el servidor deja de funcionar, aunque sea por un período breve, nadie puede colaborar ni registrar los cambios en los que esté trabajando. Asimismo, si el disco donde se almacena la base de datos central sufre daños y no existen copias de seguridad adecuadas, puede perderse todo el historial del proyecto, con excepción de las versiones que cada persona conserve en su equipo local.

Figura 3. Esquema de un sistema de control de versiones centralizado



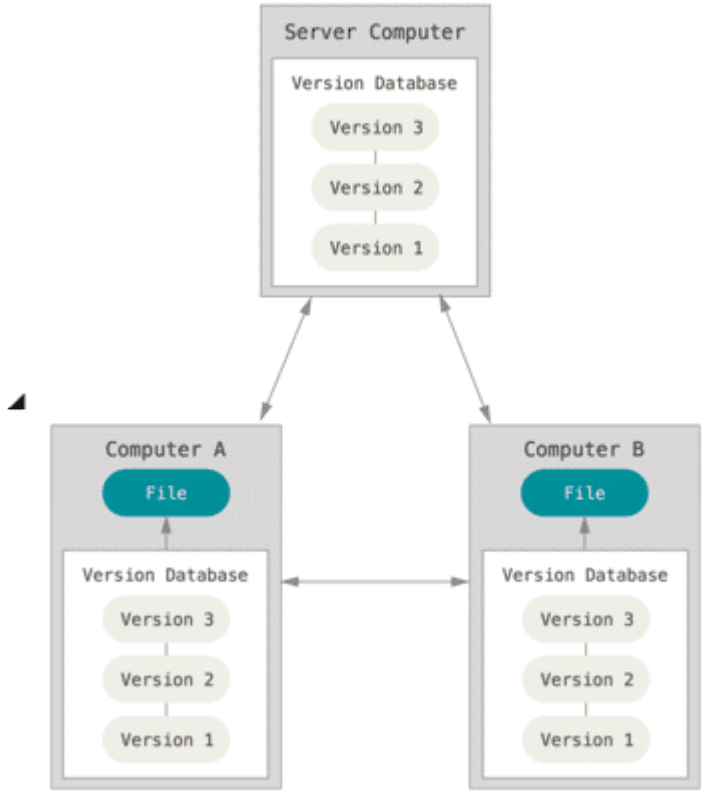
Fuente: Git-scm, s.f., https://goo.su/4ZbTQ

En los sistemas de control de versiones distribuidos, los clientes no solo descargan la última versión de los archivos, sino que replican por completo el repositorio, incluido todo su historial.

De este modo, si un servidor deja de funcionar y el equipo estaba colaborando a través de él, cualquiera de los repositorios locales puede copiarse nuevamente en el servidor para restablecer el sistema. Cada clonación constituye, en la práctica, una copia íntegra de los datos.

Además, muchos de estos sistemas permiten trabajar con varios repositorios remotos al mismo tiempo, lo que facilita la colaboración simultánea con distintos grupos dentro de un mismo proyecto. Esta característica posibilita la implementación de diversos flujos de trabajo que no pueden aplicarse en los sistemas centralizados, como los modelos jerárquicos.

Figura 4. Esquema de un sistema de control de versiones distribuido



Fuente: Git-scm, s.f.a., https://goo.su/jarE

CONTINUAR

2. Ramificaciones Git

A diferencia de otros sistemas de control de versiones, Git promueve un ciclo de desarrollo en el que las ramas se crean y se fusionan con frecuencia, incluso varias veces en un mismo día. Comprender y gestionar esta dinámica permite modificar la forma en que se desarrolla *software*, ya que facilita el trabajo paralelo y la integración continua de cambios. A continuación, se muestra un ejemplo básico de creación de una confirmación de cambios:

```
$ git add README test.rb LICENSE
```

```
$ git commit -m "initial commit of my project"
```

¿Qué es una rama?

Para comprender realmente cómo funciona la ramificación en Git, es necesario examinar primero la forma en que

almacena sus datos. Git no los guarda de manera incremental —es decir, registrando únicamente las diferencias—, sino como una serie de instantáneas que reflejan el estado completo de los archivos en un momento determinado.

En cada confirmación de cambios (*commit*), Git almacena una instantánea del trabajo preparado. Esta incluye metadatos, como el autor y el mensaje descriptivo, así como uno o varios apuntadores a las confirmaciones que constituyen sus padres directos: un solo padre en una confirmación normal y múltiples padres cuando se confirma una fusión (*merge*) de dos o más ramas.

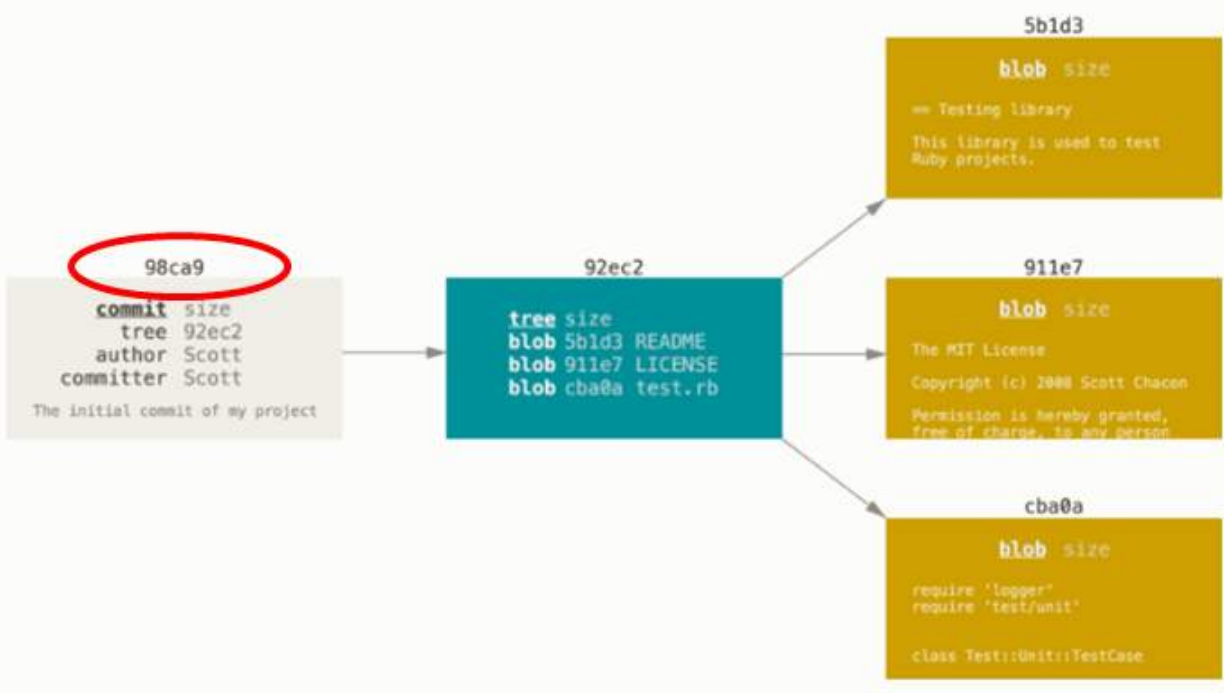
A modo de ejemplo, supóngase una carpeta con tres archivos que se preparan (*stage*) y luego se confirman (*commit*). Al preparar los archivos, Git calcula un *hash* (*sha-1*) de cada uno, guarda una copia en el repositorio —denominada «blob»— y registra cada *hash* en el área de preparación (*staging area*).

Cuando se ejecuta el comando `git commit`, Git calcula un hash de cada subdirectorio —en este caso, solo el directorio principal del proyecto— y los almacena como objetos «árbol» dentro del repositorio. Posteriormente, crea un objeto de

confirmación con los metadatos correspondientes y un apuntador al objeto árbol raíz del proyecto.

En ese momento, el repositorio contiene cinco objetos: un «blob» por cada uno de los tres archivos, un objeto «árbol» con la lista de contenidos del directorio y sus relaciones con los «blobs», y una confirmación de cambios (*commit*) que apunta a la raíz de ese árbol e incluye los metadatos pertinentes.

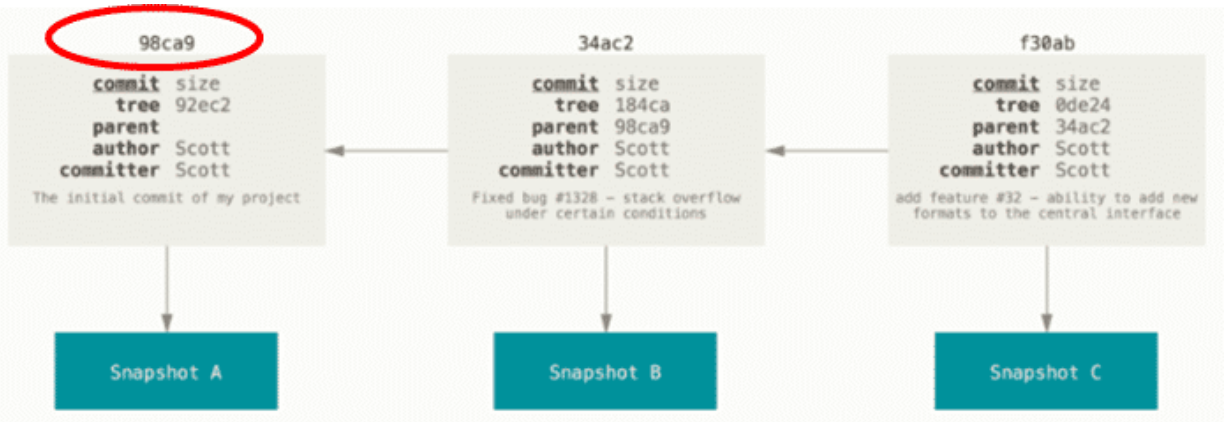
Figura 5. Representación de los objetos internos generados en una confirmación de Git



Fuente: Git-scm, s.f.b., https://goo.su/JJpMub

Si se realizan más cambios y se vuelven a confirmar, la nueva confirmación guardará un apuntador a la confirmación anterior.

Figura 6. Secuencia de confirmaciones y apuntadores entre instantáneas en Git

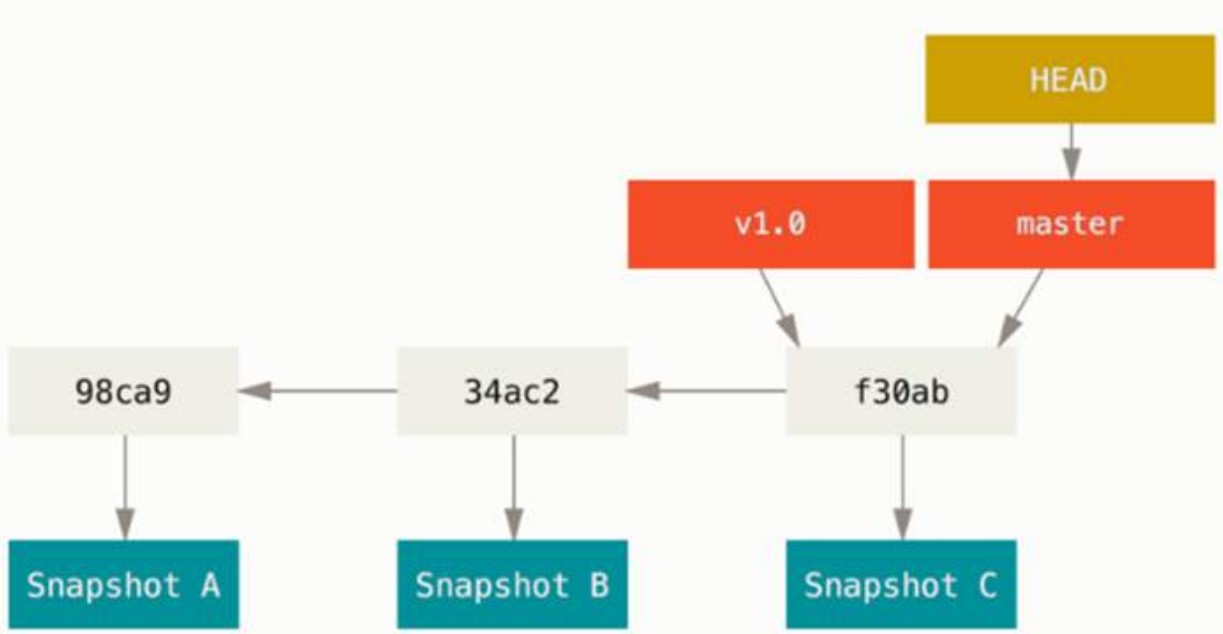


Fuente: Git-scm, s.f.b., https://goo.su/JJpMub

Una rama en Git es, simplemente, un apuntador móvil que señala una determinada confirmación. La rama predeterminada es *master*. Con la primera confirmación que se realiza, se crea esta rama principal apuntando a

dicha confirmación. A medida que se efectúan nuevas confirmaciones, la rama avanza automáticamente hacia la más reciente.

Figura 7. Una rama y su historial de confirmaciones.



Fuente: Git-scm, s.f.b., https://goo.su/JJpMub

La rama *master* no es una rama especial; funciona igual que cualquier otra. Su presencia en la mayoría de los repositorios se debe a que es la que crea por defecto el comando *git init*, y habitualmente no se modifica su nombre.

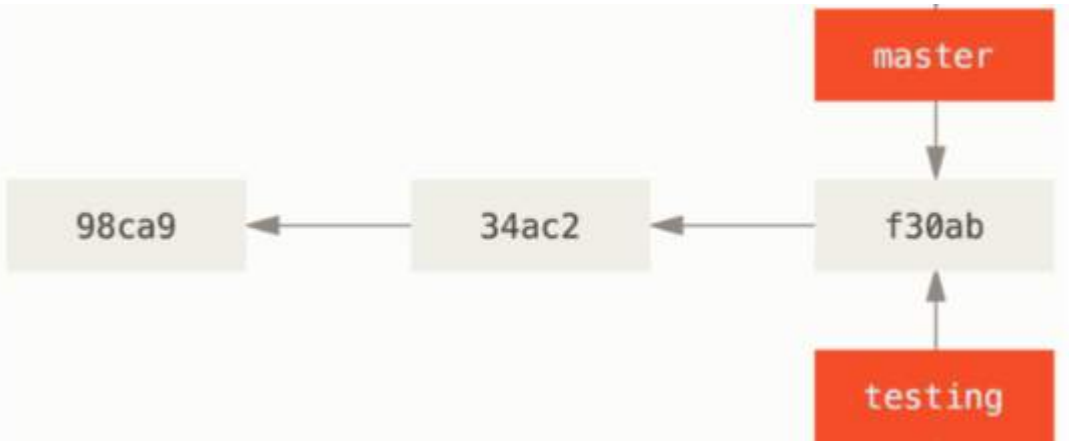
Crear una rama nueva

Cuando se crea una nueva rama, se genera un nuevo apuntador que puede desplazarse libremente. En el siguiente ejemplo, se crea una rama denominada *testing* mediante el comando:

```
$ git branch testing
```

Este comando crea un nuevo apuntador que señala la misma confirmación en la que se encuentra el repositorio en ese momento. De este modo, pasan a existir dos ramas que apuntan al mismo conjunto de confirmaciones.

Figura 8. Creación de una nueva rama que apunta a la misma confirmación actual



Fuente: Git-scm, s.f.b., https://goo.su/JJpMub

Mediante un apuntador especial denominado HEAD, Git sabe en qué rama se encuentra el repositorio en cada momento. Este apuntador señala la rama local activa; en el ejemplo, la rama *master*.

El comando git branch únicamente crea una nueva rama, pero no cambia a ella. Por lo tanto, *HEAD* continúa apuntando a la rama en la que se está trabajando actualmente.

Figura 9. Apuntador HEAD a la rama activa

Fuente: Git-scm, s.f.b., https://goo.su/JJpMub

Esto también puede comprobarse mediante la ejecución del comando `git log`, utilizando la opción `--decorate`, que permite visualizar a qué confirmación apunta cada rama:

```
$ 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
```

En la salida se observa que las ramas `master` y `testing` aparecen asociadas a la confirmación `f30ab`, lo que indica que ambas apuntan a ese mismo estado del proyecto.

Cambiar de rama

Para cambiar de una rama a otra, se utiliza el comando *git checkout*. Por ejemplo, para cambiar a la rama *testing*, recién creada, se ejecuta:

```
$ git checkout testing
```

Este comando mueve el apuntador `HEAD` hacia la rama `testing`, lo que indica que, a partir de ese momento, el trabajo se realizará sobre esa rama.

Figura 10. Movimiento del apuntador `HEAD` al cambiar de rama

Fuente: Git-scm, s.f.b., https://goo.su/JJpMub

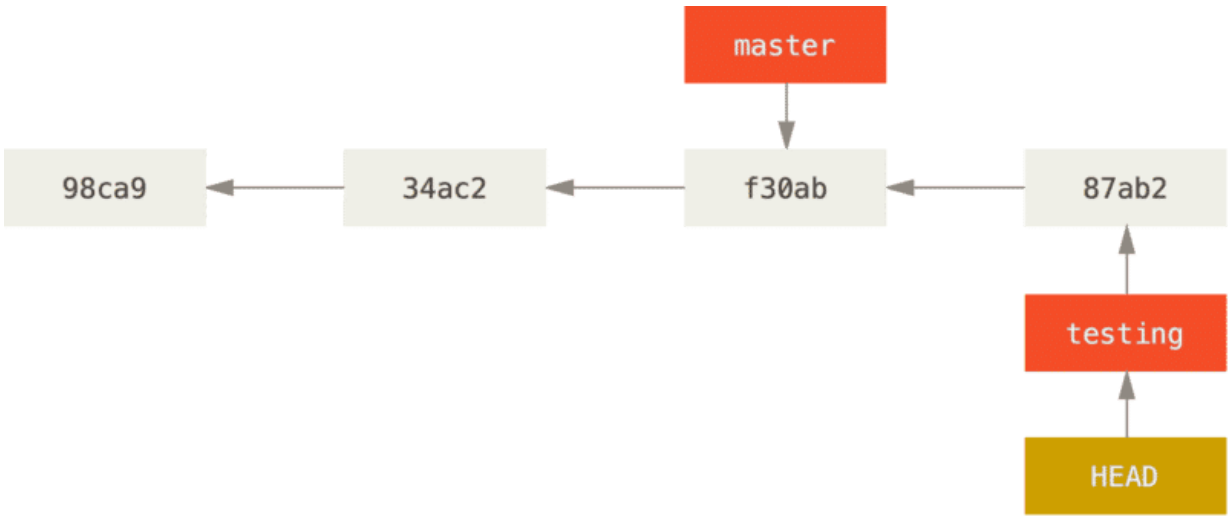
Para comprender el efecto de esta acción, supóngase que se crea un archivo nuevo llamado «test.rb» y se confirman los cambios con el siguiente comando:

```
$ git commit -a -m 'made a change'
```

Al realizar esta confirmación, la rama activa —en este caso, *testing*— avanza hacia la nueva confirmación, mientras que la rama *master*

permanece apuntando a la confirmación anterior. De este modo, ambas ramas comienzan a divergir, ya que cada una señala un estado distinto del proyecto.

Figura 11. Divergencia de ramas tras una nueva confirmación en la rama activa



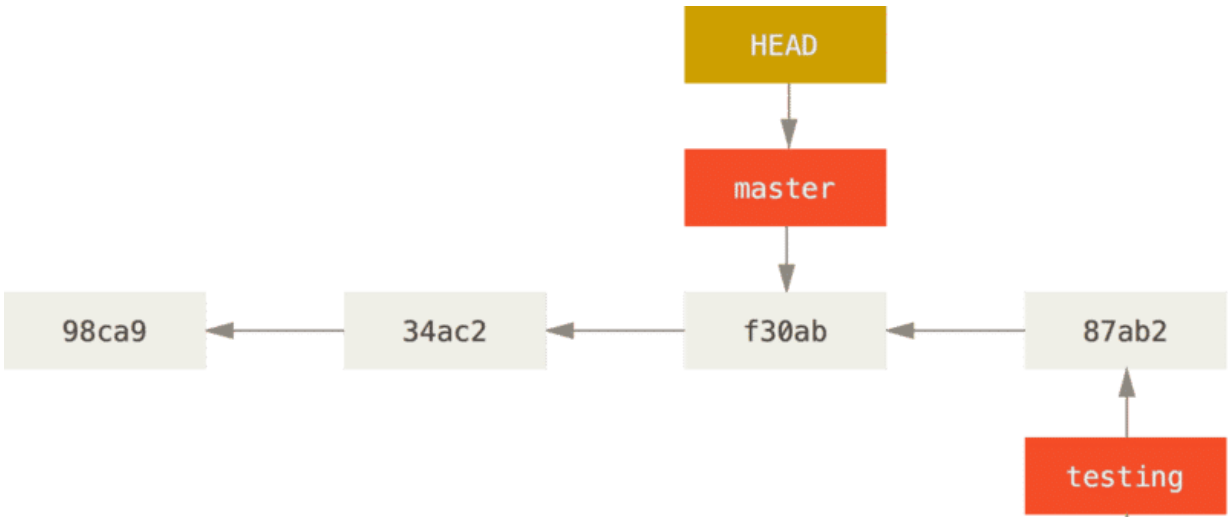
Fuente: Git-scm, s.f.b., https://goo.su/JJpMub

Aquí se observa un aspecto relevante: la rama `testing` avanza, mientras que la rama `master` permanece en la confirmación en la que se encontraba cuando se ejecutó el comando `git`

checkout para cambiar de rama. A continuación, se muestra cómo cambiar de nuevo a la rama master:

```
$ git checkout master
```

Figura 12. HEAD apunta a otra rama al cambiar de rama



Fuente: Git-scm, s.f.b., https://goo.su/JJpMub

Este comando realiza dos acciones. Por un lado, mueve el apuntador HEAD nuevamente hacia la rama master; por otro, actualiza los archivos del directorio de trabajo, dejándolos en el estado correspondiente a la última

instantánea confirmada en esa rama. A partir de ese momento, los cambios que se realicen divergirán de la versión desarrollada en la rama *testing*. En términos prácticos, se deja en pausa el trabajo realizado en esa rama para continuar el desarrollo en otra dirección.

CONTINUAR

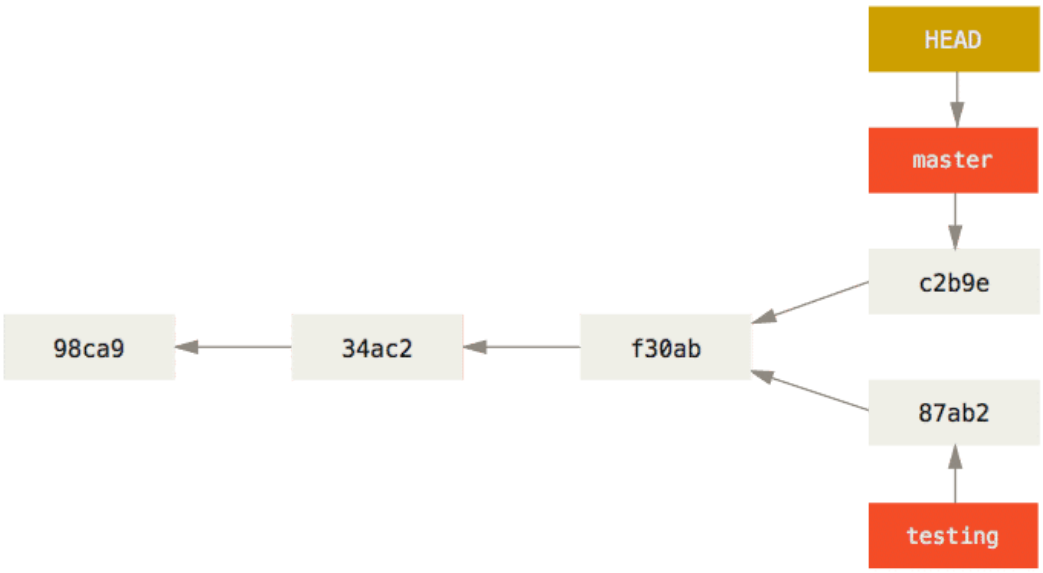
3. Cambio de archivos al saltar entre ramas

Al cambiar de rama en Git, los archivos del directorio de trabajo se modifican automáticamente. Si se cambia a una rama anterior, el directorio adopta el estado que tenía en la última confirmación de esa rama. Si Git no puede efectuar el cambio de manera limpia —por ejemplo, debido a modificaciones sin confirmar—, no permitirá realizar el cambio.

Si se realizan nuevos cambios, por ejemplo en el mismo archivo «test.rb», y luego se confirman con el siguiente comando `$ git commit -a -m 'made other changes'`, el historial del proyecto comienza a divergir. Se ha creado una rama y se ha trabajado en ella; luego se ha regresado a la rama original y también se ha trabajado allí. Los cambios efectuados en cada línea de desarrollo permanecen aislados en ramas independientes, entre las cuales es posible alternar

libremente. Todo ello se logra mediante tres comandos: `git branch`, `git checkout` y `git commit`.

Figura 13. Divergencia del historial tras realizar cambios en ramas distintas



Fuente: Git-scm, s.f.b., https://goo.su/JJpMub

Esta estructura puede visualizarse con el siguiente comando:

`$ 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
```

Una rama en Git es, en esencia, un archivo que contiene los 40 caracteres de una suma de comprobación *sha-1*, que representan la confirmación a la que apunta. Esto contrasta con otros sistemas de control de versiones, en los que crear una nueva rama implica copiar todos los archivos del proyecto en un directorio adicional. Ese proceso puede demorar segundos o minutos, según el tamaño del proyecto, mientras que en Git la creación de ramas es inmediata.

Además, como cada confirmación registra sus nodos padre, la identificación de la base adecuada para fusionar ramas es un procedimiento automático y, por lo general, sencillo. Esto favorece el uso frecuente de ramificaciones dentro del flujo de trabajo.

CONTINUAR

4. Ramificaciones en Git: ramificar y fusionar

A continuación, se presenta un ejemplo sencillo de ramificación y fusión que refleja un posible flujo de trabajo en un entorno real. Imaginemos la siguiente situación:

- se trabaja en un sitio web;
- se crea una rama para desarrollar una nueva funcionalidad;
- se realizan cambios en esa rama.

En ese momento, surge un problema crítico que debe resolverse con urgencia. El procedimiento podría desarrollarse del siguiente modo:

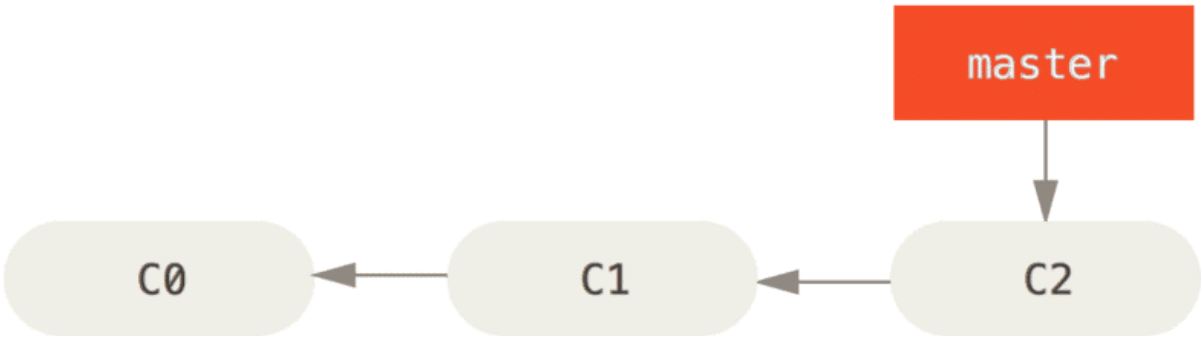
- se regresa a la rama principal de producción;

- se crea una nueva rama destinada a solucionar el problema detectado;
- se trabaja en esa rama hasta resolverlo;
- tras realizar las pruebas correspondientes, se fusiona (*merge*) esa rama y se envía (*push*) a la rama de producción;
- finalmente, se vuelve a la rama en la que se estaba trabajando antes de la interrupción y se continúa con el desarrollo pendiente.

Procedimientos básicos de ramificación

Imaginemos que estamos trabajando en un proyecto y que ya hemos realizado varias confirmaciones (*commit*).

Figura 14. Historial inicial con varias confirmaciones en la rama *master*



Fuente: Git-scm, s.f.c., https://goo.su/rBCbT

Decidimos trabajar en el problema #53, de acuerdo con el sistema de seguimiento utilizado por la organización. Para crear una nueva rama y cambiar a ella en un solo paso, puede emplearse el comando `git checkout` con la opción `-b`:

```
$ git checkout -b iss53
```

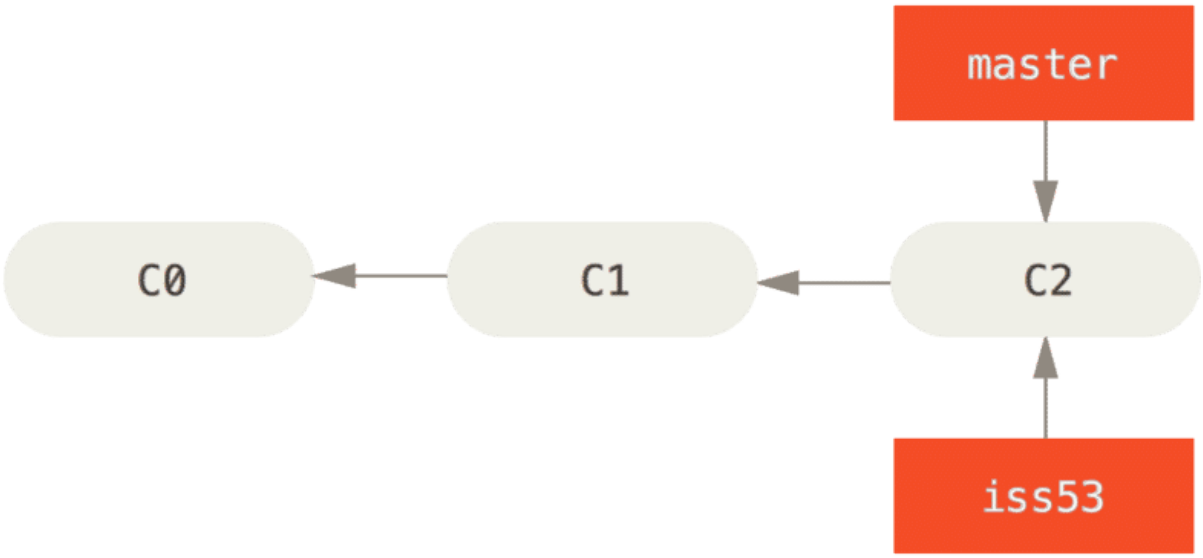
```
Switched to a new branch "iss53"
```

Este comando equivale a ejecutar de forma consecutiva:

```
$ git branch iss53
```

```
$ git checkout iss53
```

Figura 15. Creación de la rama `iss53` y cambio a dicha rama



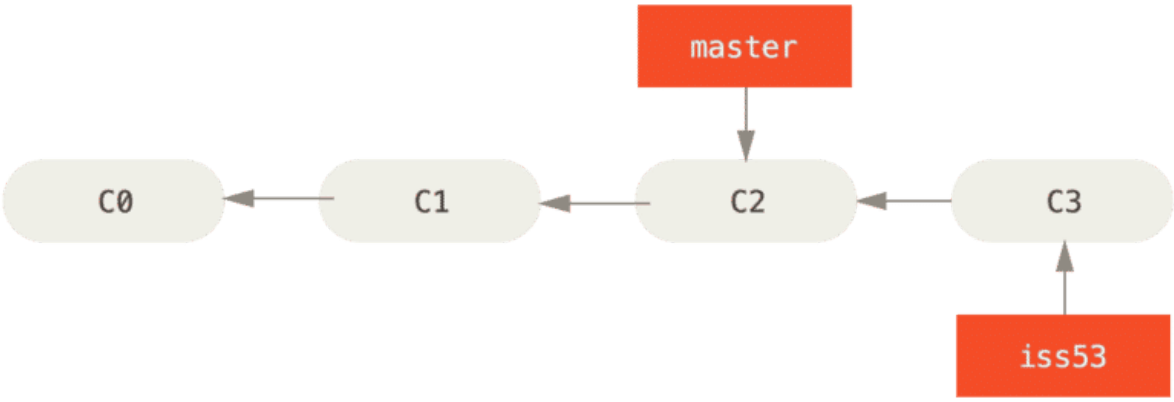
Fuente: Git-scm, s.f.c., https://goo.su/rBCbT

Trabajamos en el sitio web —por ejemplo, en el archivo index.html— y realizamos algunas confirmaciones (*commit*). Con ello, avanza la rama iss53, que es la que se encuentra activa en ese momento (es decir, a la que apunta HEAD):

$ vim index.html (Modifico el archivo index.html usando editor texto en Linux)

```
$ git commit -a -m 'added a new footer [issue 53]'
```

Figura 16. La rama iss53 ha avanzado con el trabajo realizado



Fuente: Git-scm, s.f.c., https://goo.su/rBCbT

En ese momento, se recibe una notificación sobre otro problema urgente en el sitio web que debe resolverse de inmediato. Con Git, no es necesario mezclar este nuevo inconveniente con los cambios ya realizados para el problema #53, ni perder tiempo revirtiéndolos para poder trabajar sobre la versión en producción. Basta con volver a la rama `master` y continuar desde allí.

No obstante, antes de hacerlo, debe considerarse que si existen cambios sin confirmar en el directorio de trabajo o en

el área de preparación (*staging area*), Git no permitirá cambiar de rama si pueden generarse conflictos. Por ello, es recomendable mantener un estado de trabajo limpio antes de alternar entre ramas. Para lograrlo, existen procedimientos como el *stashing* o la modificación de confirmaciones.

En este caso, como todos los cambios han sido confirmados, es posible cambiar a la rama *master* sin inconvenientes:

```
$ git checkout master
```

```
Switched to branch 'master
```

Tras ejecutar este comando, el directorio de trabajo queda exactamente en el estado en que se encontraba antes de comenzar a trabajar en el problema #53, lo que permite concentrarse en el nuevo inconveniente.

Es importante recordar que Git actualiza el directorio de trabajo al estado correspondiente a la confirmación señalada por la rama que se activa mediante checkout. El sistema agrega,

elimina o modifica archivos automáticamente para que la copia de trabajo refleje con exactitud la última confirmación realizada en esa rama.

A continuación, se procede a resolver el problema urgente. Para ello, se crea una nueva rama denominada hotfix, sobre la cual se trabajará hasta solucionarlo:

```
$ git checkout -b hotfix
```

```
Switched to a new branch 'hotfix'
```

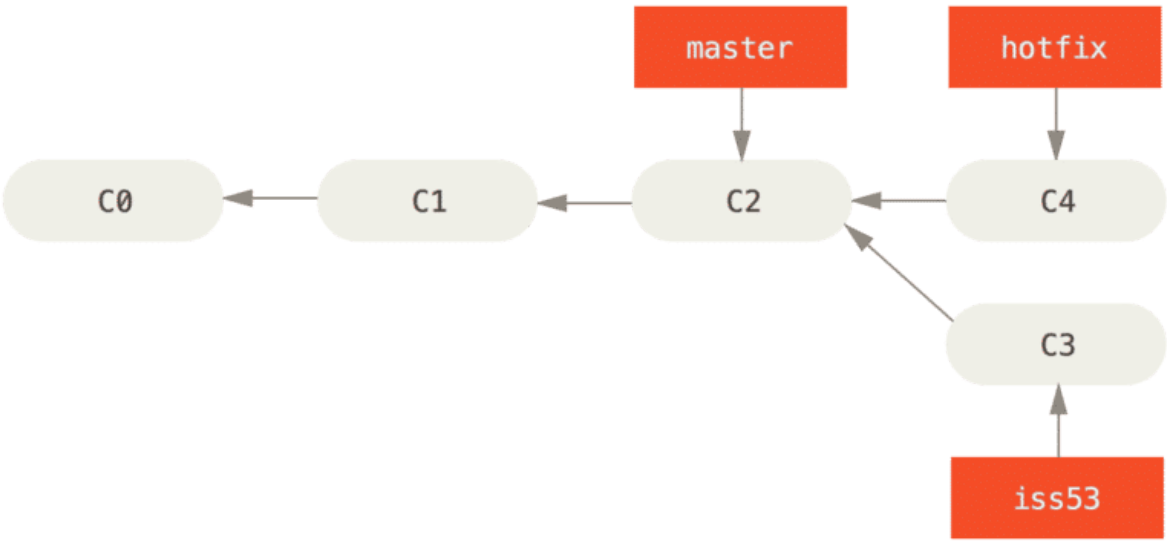
```
$ vim index.html (modificamos el archivo index.html usando sistema operativo Linux)
```

```
$ git commit -a -m 'fixed the broken email address'
```

```
[hotfix 1fb7853] fixed the broken email address
```

```
 1 file changed, 2 insertions(+)
```

Figura 17. Creación de la rama hotfix y nueva confirmación para resolver el problema urgente



Fuente: Git-scm, s.f.c., https://goo.su/rBCbT

Se pueden realizar las pruebas necesarias, verificar que la solución sea correcta e incorporar los cambios a la rama `master` para su puesta en producción. Esto se lleva a cabo con el comando `git merge`:

```
$ git checkout master
```

```
$ git merge hotfix
```

```
Updating f42c576..3a0874c
```

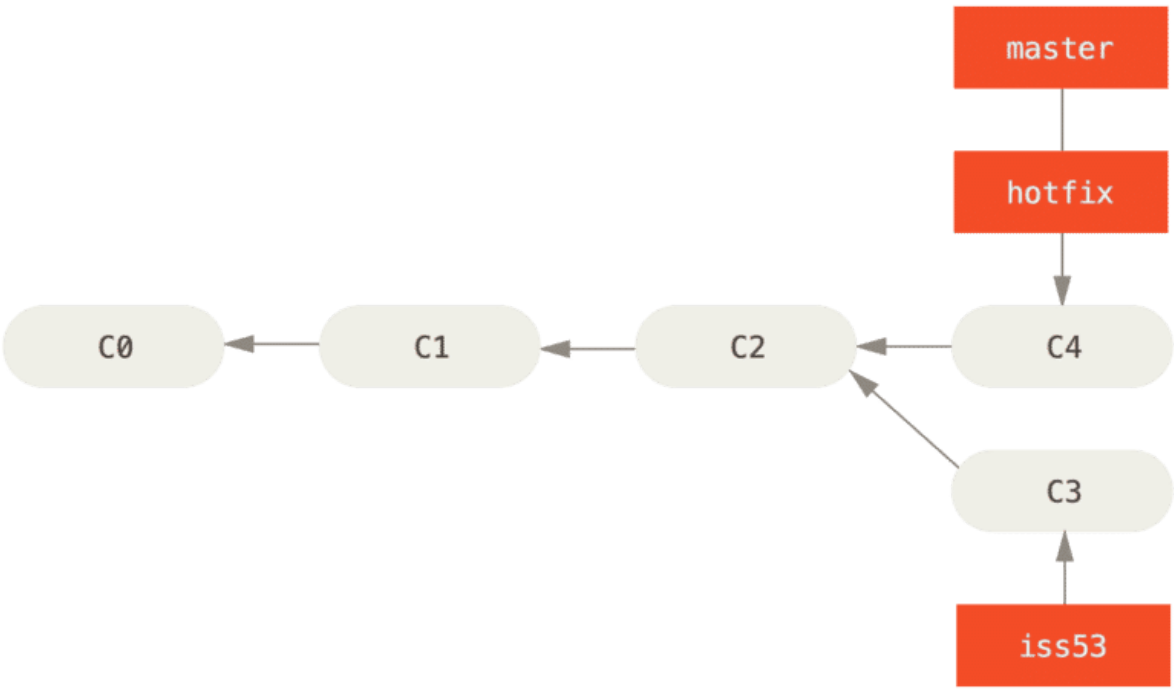
```
Fast-forward

 index.html | 2 ++

 1 file changed, 2 insertions(+)
```

En la salida del comando aparece la expresión `fast-forward`. Esto indica que Git ha avanzado el apuntador de la rama, ya que la confirmación de la rama `hotfix` se encontraba directamente por delante en el historial de la rama *master*.

En otras palabras, cuando se fusiona una confirmación que es descendiente directa de la confirmación actual, Git no necesita crear una nueva confirmación de fusión, sino que simplemente mueve el apuntador hacia adelante. Al no existir trabajo divergente que integrar, el proceso se simplifica mediante este «avance rápido».

De este modo, los cambios realizados quedan incorporados en la instantánea correspondiente a la confirmación señalada por la rama master, y pueden desplegarse en producción.

Figura 18. Fusión mediante fast-forward de la rama hotfix en master



Fuente: Git-scm, s.f.c., https://goo.su/rBCbT

Una vez resuelto el problema urgente que había interrumpido el trabajo, es posible retomarlo. Antes de hacerlo, conviene eliminar la rama hotfix, ya que ya no es

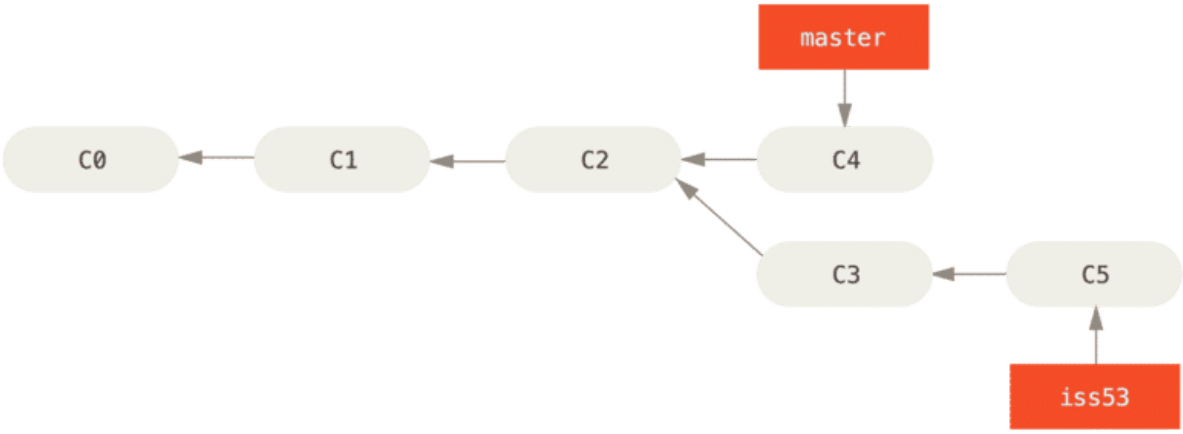
necesaria, dado que apunta al mismo lugar que la rama master. Esto puede realizarse con la opción -d del comando git branch:

```
$ git branch -d hotfix

Deleted branch hotfix (3a0874c).
```

Con esto, todo queda 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(+)
```

Figura 19. Continuación del trabajo en la rama iss53 tras eliminar la rama hotfix



Fuente: Git-scm, s.f.c., https://goo.su/rBCbT

Cabe señalar que los cambios realizados en la rama hotfix no están presentes en los archivos de la rama iss53. Si fuera necesario incorporarlos, podría fusionarse (*merge*) la rama master en la rama iss53 mediante el comando git merge master. También es posible esperar y efectuar la fusión cuando se decida integrar la rama iss53 en la rama master.

Procedimientos básicos de fusión

Supongamos que el trabajo relacionado con el problema #53 ya está completo y listo para fusionarse (*merge*) con la rama *master*. Para ello, de manera similar a lo realizado

anteriormente con la rama hotfix, se procede a fusionar la rama iss53.

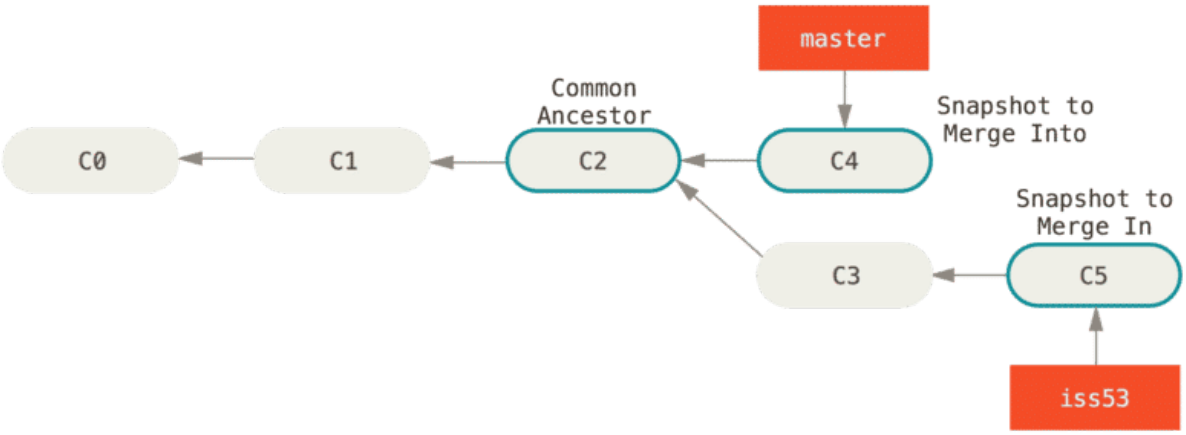
Primero, se activa (checkout) la rama en la que se desea integrar los cambios y luego se ejecuta el comando git merge:

```
$ git checkout master

Switched to branch 'master'

$ git merge iss53

Merge made by the 'recursive' strategy.

index.html |    1 +

1 file changed, 1 insertion(+)
```

Este proceso difiere del realizado anteriormente con hotfix. En este caso, el historial de desarrollo había divergido en un punto previo. Como la confirmación de la rama actual no es ancestro directo de la rama que se desea fusionar, Git debe realizar un procedimiento adicional.

En estas circunstancias, Git efectúa una fusión de tres vías, utilizando las dos instantáneas correspondientes a los extremos de cada rama y la instantánea del ancestro común a ambas.

Figura 20. Fusión de tres vías utilizando el ancestro común de ambas ramas



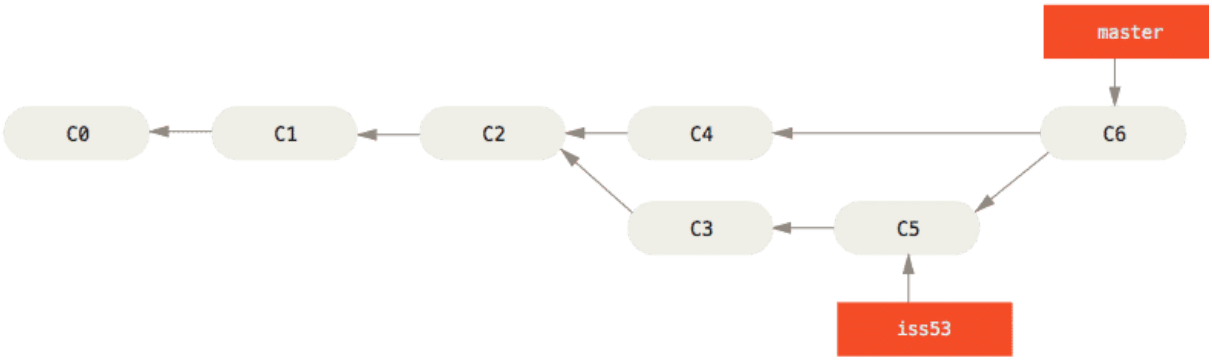
Fuente: Git-scm, s.f.c., https://goo.su/rBCbT

En lugar de limitarse a avanzar el apuntador de la rama, Git crea una nueva instantánea resultante de la fusión de tres vías y genera

automáticamente una nueva confirmación (*commit*) que apunta a ella.

Este proceso se denomina «fusión confirmada» y se caracteriza por tener más de un padre, ya que la nueva confirmación referencia a las dos confirmaciones provenientes de las ramas que se han fusionado.

Figura 21. Creación de una confirmación de fusión con múltiples padres



Fuente: Git-scm, s.f.c., https://goo.su/rBCbT

Conviene destacar que es el propio Git quien determina automáticamente el ancestro común más adecuado para realizar la fusión. A diferencia de otros sistemas, como CVS o Subversion, en los que el desarrollador debe identificar dicho ancestro, en Git este proceso se resuelve de manera automática, lo que facilita las fusiones.

Una vez que el trabajo ha sido integrado en la rama principal, la rama *iss53* deja de ser necesaria. Por lo tanto, puede eliminarse y cerrarse manualmente el problema en el sistema de seguimiento correspondiente:

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

Principales conflictos que pueden surgir en las fusiones

En determinadas ocasiones, los procesos de fusión no resultan automáticos. Si existen modificaciones distintas en una misma sección de un archivo en las dos ramas que se intenta fusionar, Git no podrá integrarlas directamente.

Por ejemplo, si en el trabajo del problema #53 se modificó la misma parte del archivo que también fue alterada en la rama *hotfix*, aparecerá un conflicto similar al siguiente:

```
$ git merge iss53

Auto-merging index.html

CONFLICT (content): Merge conflict in index.html

Automatic merge failed; fix conflicts and then commit the result.
```

En este caso, Git no crea automáticamente una confirmación de fusión (*merge commit*), sino que detiene el proceso hasta que se resuelva el conflicto de manera manual. Para comprobar qué archivos permanecen sin fusionar, puede utilizarse 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")
```

Los archivos conflictivos se marcan como «sin fusionar» (*unmerged*). Git añade en ellos marcadores especiales que orientan durante la resolución manual. El contenido puede verse, por ejemplo, de la siguiente manera:

```
<<<<<<< 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
```

En este bloque se indica que la versión correspondiente a HEAD —es decir, la rama activa antes de ejecutar la fusión— aparece en la parte superior (por encima de =======), mientras que la versión de la rama iss53 figura en la parte inferior. Para resolver el conflicto, debe seleccionarse manualmente el contenido adecuado o combinar ambas versiones. Por ejemplo:

```
<div id="footer">

please contact us at email.support@github.com

</div>
```

En esta solución se han integrado elementos de ambas versiones y se han eliminado los marcadores `<<<<<<<`, `=======` y `>>>>>>>`.

Una vez resueltos todos los conflictos, se deben ejecutar comandos *git add* para marcar los archivos como

preparados (*staged*), lo que indica a Git que el conflicto ha sido solucionado.

Si en lugar de editar manualmente se prefiere utilizar una herramienta gráfica, puede emplearse el comando *git mergetool*, que abrirá una herramienta de resolución de conflictos:

```
$ 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)

Al cerrar la herramienta de fusión, Git consultará si los conflictos han sido resueltos y si la fusión puede completarse. Si se confirma que así es, el sistema marcará como preparado (*staged*) el archivo que acaba de modificarse.

En cualquier momento puede ejecutarse el comando *git status* para verificar si todos los conflictos han sido solucionados:

$ 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 se ha resuelto correctamente y los archivos conflictivos aparecen como preparados, puede ejecutarse el comando `git commit` para finalizar la fusión. El mensaje de confirmación predeterminado será similar al siguiente:

```
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

#

Este mensaje puede modificarse para añadir información adicional sobre la forma en que se resolvió la fusión, si se considera pertinente para facilitar su comprensión en el futuro.

Si deseas investigar sobre fusiones avanzadas, puedes leer la siguiente publicación:

Fuente: Git-scm, (s.f.d.). *Herramientas de Git - Fusión Avanzada*. https://git-scm.com/book/es/v2/Herramientas-de-Git-Fusi%c3%b3n-Avanzada#r_advanced_merging

CONTINUAR

Referencias

GitHub, (s.f.). *Guía de mensajes de commit.* https://github.com/RomuloOliveira/commit-messages-guide/blob/master/README_es-AR.md

Git-scm, (s.f.). *Distributed Git - Distributed Workflows.* https://git-scm.com/book/id/v2/Distributed-Git-Distributed-Workflows

Git-scm, (s.f.a.). *Inicio - Sobre el Control de Versiones - Acerca del Control de Versiones.* https://git-scm.com/book/es/v2/Inicio---Sobre-el-Control-de-Versiones-Acerca-del-Control-de-Versiones

Git-scm, (s.f.b.). *Ramificaciones en Git - ¿Qué es una rama?* https://git-scm.com/book/es/v2/Ramificaciones-en-Git-%C2%BFQu%C3%A9-es-una-rama%3F

Git-scm, (s.f.c.). *Ramificaciones en Git - Procedimientos Básicos para Ramificar y Fusionar.* https://git-scm.com/book/es/v2/Ramificaciones-en-Git-Procedimientos-B%C3%A1sicos-para-Ramificar-y-Fusionar

Git-scm, (s.f.d.). *Herramientas de Git - Fusión Avanzada.* https://git-scm.com/book/es/v2/Herramientas-de-Git-Fusi%c3%b3n-Avanzada#r_advanced_merging

Referencias bibliográficas de consulta

Atlassian, (s.f.). *Git merge.* https://www.atlassian.com/es/git/tutorials/using-branches/git-merge

Atlassian, (s.f.a.). *Opciones y ejemplos de estrategias de git merge.* https://www.atlassian.com/es/git/tutorials/using-branches/merge-strategy

Atlassian, (s.f.b.). *La fusión frente a la reorganización.* https://www.atlassian.com/es/git/tutorials/merging-vs-rebasing

DataCamp, (2025). *Tutorial sobre cómo resolver conflictos de fusión en Git.* https://www.datacamp.com/es/tutorial/how-to-

resolve-merge-conflicts-in-git-tutorial

GitHub, (s.f.). *Acerca de los conflictos de fusión*. https://docs.github.com/es/pull-requests/collaborating-with-pull-requests/addressing-merge-conflicts/about-merge-conflicts

Developer María, (s.f.). F*lujo de trabajo eficiente en Git: de la idea al despliegue*. https://dev.to/marmariadev/flujo-de-trabajo-eficiente-en-git-de-la-idea-al-despliegue-23bd

Microsoft, (s.f.). *Resolve merge conflicts*. https://learn.microsoft.com/en-us/azure/devops/repos/git/merging?view=azure-devops&tabs=visual-studio-2022

CONTINUAR