Introducción a GIT

Acerca de lo que es el “Control de versiones”

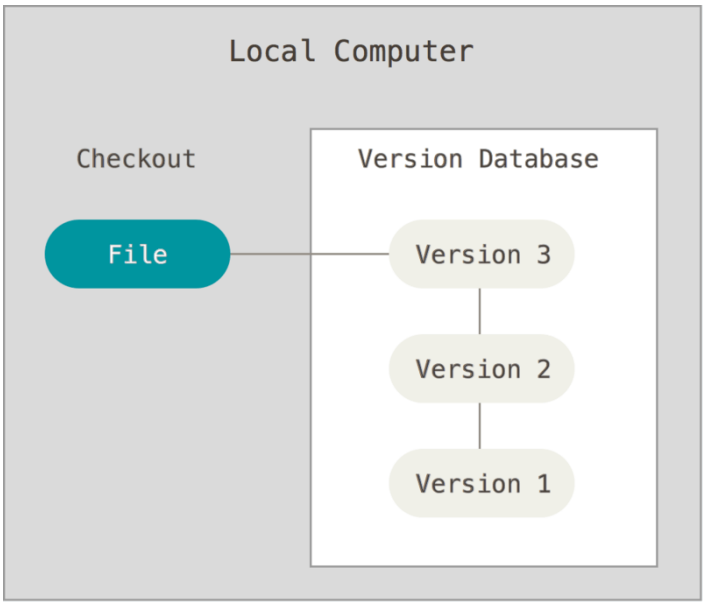
¿Qué es un control de versiones, y por qué debería importarte? Un control de versiones es un sistema que registra los cambios realizados en un archivo o conjunto de archivos a lo largo del tiempo, de modo que puedas recuperar versiones específicas más adelante. Aunque en los ejemplos de este libro usarás archivos de código fuente como aquellos cuya versión está siendo controlada, en realidad puedes hacer lo mismo con casi cualquier tipo de archivo que encuentres en una computadora.

Si eres diseñador gráfico o de web y quieres mantener cada versión de una imagen o diseño (es algo que sin duda vas a querer), usar un sistema de control de versiones (VCS por sus siglas en inglés) es una decisión muy acertada. Dicho sistema te permite regresar a versiones anteriores de tus archivos, regresar a una versión anterior del proyecto completo, comparar cambios a lo largo del tiempo, ver quién modificó por última vez algo que pueda estar causando problemas, ver quién introdujo un problema y cuándo, y mucho más. Usar un VCS también significa generalmente que si arruinas o pierdes archivos, será posible recuperarlos fácilmente. Adicionalmente, obtendrás todos estos beneficios a un costo muy bajo.

Sistemas de Control de Versiones Locales

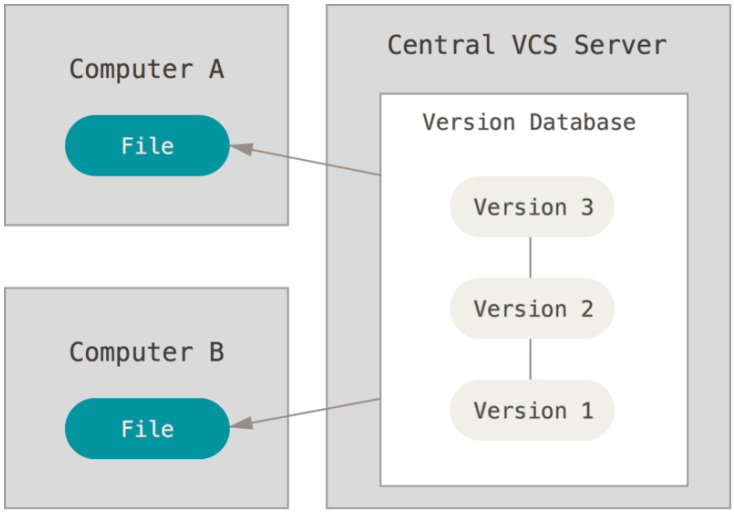
Un método de control de versiones, usado por muchas personas, es copiar los archivos a otro directorio (quizás indicando la fecha y hora en que lo hicieron, si son ingeniosos). Este método es muy común porque es muy sencillo, pero también es tremendamente propenso a errores. Es fácil olvidar en qué directorio te encuentras y guardar accidentalmente en el archivo equivocado o sobrescribir archivos que no querías.

Para afrontar este problema los programadores desarrollaron hace tiempo VCS locales que contenían una simple base de datos, en la que se llevaba el registro de todos los cambios realizados a los archivos.



Sistemas de Control de Versiones Centralizados

El siguiente gran problema con el que se encuentran las personas es que necesitan colaborar con desarrolladores en otros sistemas. Los sistemas de Control de Versiones Centralizados (CVCS por sus siglas en inglés) fueron desarrollados para solucionar este problema. Estos sistemas, como CVS, Subversion y Perforce, tienen un único servidor que contiene todos los archivos versionados y varios clientes que descargan los archivos desde ese lugar central. Este ha sido el estándar para el control de versiones por muchos años.

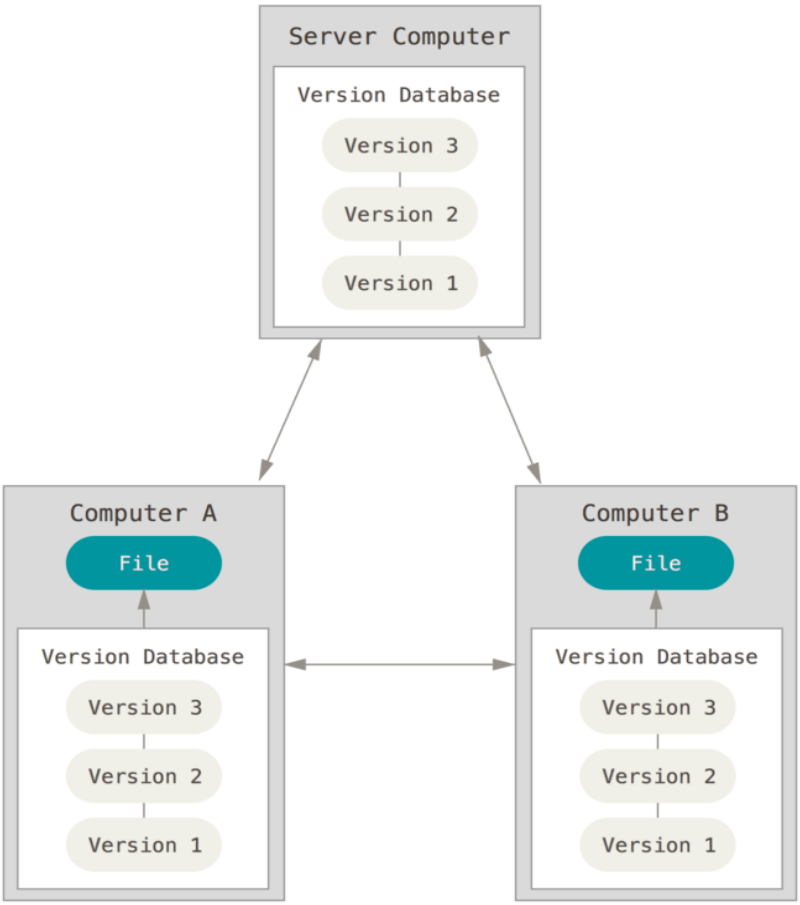


Esta configuración ofrece muchas ventajas, especialmente frente a VCS locales. Por ejemplo, todas las personas saben hasta cierto punto en qué están trabajando los otros colaboradores del proyecto. Los administradores tienen control detallado sobre qué puede hacer cada usuario, y es mucho más fácil administrar un CVCS que tener que lidiar con bases de datos locales en cada cliente.

Sin embargo, esta configuración también tiene serias desventajas. La más obvia es el punto único de fallo que representa el servidor centralizado. Si ese servidor se cae durante una hora, entonces durante esa hora nadie podrá colaborar o guardar cambios en archivos en los que hayan estado trabajando. Si el disco duro en el que se encuentra la base de datos central se corrompe, y no se han realizado copias de seguridad adecuadamente, se perderá toda la información del proyecto, con excepción de las copias instantáneas que las personas tengan en sus máquinas locales. Los VCS locales sufren de este mismo problema: Cuando tienes toda la historia del proyecto en un mismo lugar, te arriesgas a perderlo todo.

Sistemas de Control de Versiones Distribuidos

Los sistemas de Control de Versiones Distribuidos (DVCS por sus siglas en inglés) ofrecen soluciones para los problemas que han sido mencionados. En un DVCS (como Git, Mercurial, Bazaar o Darcs), los clientes no solo descargan la última copia instantánea de los archivos, sino que se replica completamente el repositorio. De esta manera, si un servidor deja de funcionar y estos sistemas estaban colaborando a través de él, cualquiera de los repositorios disponibles en los clientes puede ser copiado al servidor con el fin de restaurarlo. Cada clon es realmente una copia completa de todos los datos.



Fundamentos de Git:

Entonces, ¿qué es Git en pocas palabras? Es muy importante entender bien esta sección, porque si entiendes lo que es Git y los fundamentos de cómo funciona, probablemente te será mucho más fácil usar Git efectivamente. A medida que aprendas Git, intenta olvidar todo lo que posiblemente conoces acerca de otros VCS como Subversion y Perforce. Hacer esto te ayudará a evitar confusiones sutiles a la hora de utilizar la herramienta. Git almacena y maneja la información de forma muy diferente a esos otros sistemas, a pesar de que su interfaz de usuario es bastante similar. Comprender esas diferencias evitará que te confundas a la hora de usarlo.

Copias instantáneas, no diferencias

La principal diferencia entre Git y cualquier otro VCS (incluyendo Subversion y sus amigos) es la forma en la que manejan sus datos. Conceptualmente, la mayoría de los otros sistemas almacenan la información como una lista de cambios en los archivos. Estos sistemas (CVS, Subversion, Perforce, Bazaar, etc.) manejan la información que almacenan como un conjunto de archivos y las modificaciones hechas a cada uno de ellos a través del tiempo.

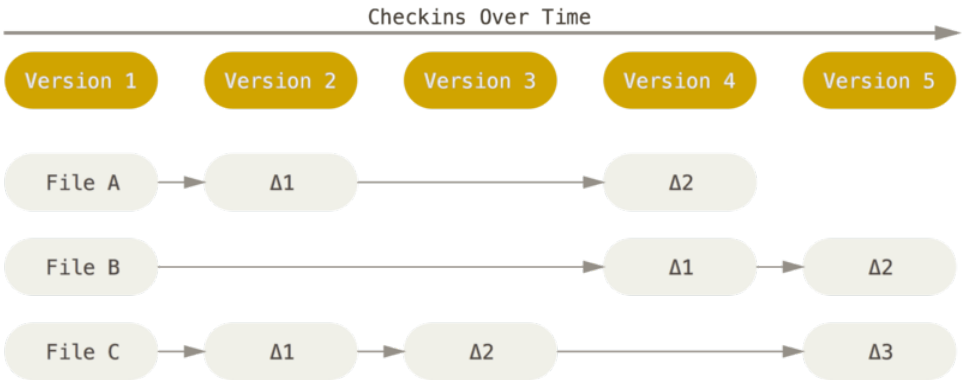


Figure 4. Almacenamiento de datos como cambios en una versión de la base de cada archivo.

Git no maneja ni almacena sus datos de esta forma. Git maneja sus datos como un conjunto de copias instantáneas de un sistema de archivos miniatura. Cada vez que confirmas un cambio, o guardas el estado de tu proyecto en Git, él básicamente toma una foto del aspecto de todos tus archivos en ese momento y guarda una referencia a esa copia instantánea. Para ser eficiente, si los archivos no se han modificado Git no almacena el archivo de nuevo, sino un enlace al archivo anterior idéntico que ya tiene almacenado. Git maneja sus datos como una secuencia de copias instantáneas.

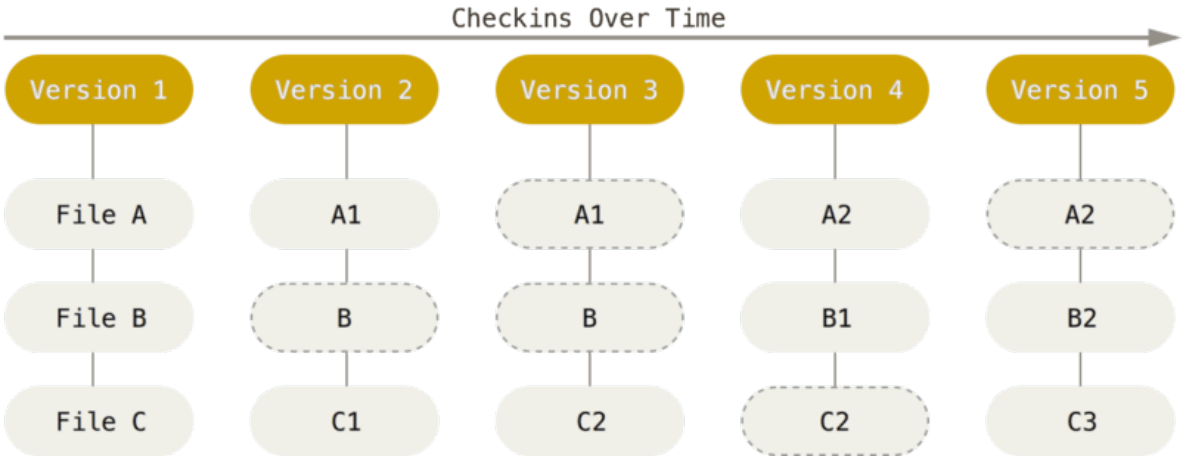


Figure 5. Almacenamiento de datos como instantáneas del proyecto a través del tiempo.

Esta es una diferencia importante entre Git y prácticamente todos los demás VCS. Hace que Git reconsidere casi todos los aspectos del control de versiones que muchos de los demás sistemas copiaron de la generación anterior. Esto hace que Git se parezca más a un sistema de archivos miniatura con algunas herramientas tremendamente poderosas desarrolladas sobre él, que a un VCS. Exploraremos algunos de los beneficios que obtienes al modelar tus datos de esta manera cuando veamos ramificación (branching) en Git en el.

Casi todas las operaciones son locales

La mayoría de las operaciones en Git sólo necesitan archivos y recursos locales para funcionar. Por lo general no se necesita información de ningún otro computador de tu red. Si estás acostumbrado a un CVCS donde la mayoría de las operaciones tienen el costo adicional del retardo de la red, este aspecto de Git te va a hacer pensar que los dioses de la velocidad han bendecido Git con poderes sobrenaturales. Debido a que tienes toda la historia del proyecto ahí mismo, en tu disco local, la mayoría de las operaciones parecen prácticamente inmediatas.

Por ejemplo, para navegar por la historia del proyecto, Git no necesita conectarse al servidor para obtener la historia y mostrártela - simplemente la lee directamente de tu base de datos local. Esto significa que ves la historia del proyecto casi instantáneamente. Si quieres ver los cambios introducidos en un archivo entre la versión actual y la de hace un mes, Git puede buscar el archivo de hace un mes y hacer un cálculo de diferencias localmente, en lugar de tener que pedirle a un servidor remoto que lo haga, u obtener una versión antigua desde la red y hacerlo de manera local.

Esto también significa que hay muy poco que no puedes hacer si estás desconectado o sin VPN. Si te subes a un avión o a un tren y quieres trabajar un poco, puedes confirmar tus cambios felizmente hasta que consigas una conexión de red para subirlos. Si te vas a casa y no consigues que tu cliente VPN funcione correctamente, puedes seguir trabajando. En muchos otros sistemas, esto es imposible o muy engorroso. En Perforce, por ejemplo, no puedes hacer mucho cuando no estás conectado al servidor. En Subversion y CVS, puedes editar archivos, pero no puedes confirmar los cambios a tu base de datos (porque tu base de datos no tiene conexión). Esto puede no parecer gran cosa, pero te sorprendería la diferencia que puede suponer.

Git tiene integridad

Todo en Git es verificado mediante una suma de comprobación (checksum en inglés) antes de ser almacenado, y es identificado a partir de ese momento mediante dicha suma. Esto significa que es imposible cambiar los contenidos de cualquier archivo o directorio sin que Git lo sepa. Esta funcionalidad está integrada en Git al más bajo nivel y es parte integral de su filosofía. No puedes perder información durante su transmisión o sufrir corrupción de archivos sin que Git sea capaz de detectarlo.

El mecanismo que usa Git para generar esta suma de comprobación se conoce como hash SHA-1. Se trata de una cadena de 40 caracteres hexadecimales (0-9 y a-f), y se calcula con base en los contenidos del archivo o estructura del directorio en Git. Un hash SHA-1 se ve de la siguiente forma:

24b9da6552252987aa493b52f8696cd6d3b00373

Verás estos valores hash por todos lados en Git, porque son usados con mucha frecuencia. De hecho, Git guarda todo no por nombre de archivo, sino por el valor hash de sus contenidos.

Git generalmente solo añade información

Cuando realizas acciones en Git, casi todas ellas sólo añaden información a la base de datos de Git. Es muy difícil conseguir que el sistema haga algo que no se pueda enmendar, o que de algún modo

borre información. Como en cualquier VCS, puedes perder o estropear cambios que no has confirmado todavía. Pero después de confirmar una copia instantánea en Git es muy difícil perderla, especialmente si envías tu base de datos a otro repositorio con regularidad.

Esto hace que usar Git sea un placer, porque sabemos que podemos experimentar sin peligro de estropear gravemente las cosas. Para un análisis más exhaustivo de cómo almacena Git su información y cómo puedes recuperar datos aparentemente perdidos.

Los Tres Estados

Ahora presta atención. Esto es lo más importante que debes recordar acerca de Git si quieres que el resto de tu proceso de aprendizaje prosiga sin problemas. Git tiene tres estados principales en los que se pueden encontrar tus archivos: confirmado (committed), modificado (modified), y preparado (staged). Confirmado: significa que los datos están almacenados de manera segura en tu base de datos local. Modificado: significa que has modificado el archivo, pero todavía no lo has confirmado a tu base de datos. Preparado: significa que has marcado un archivo modificado en su versión actual para que vaya en tu próxima confirmación.

Esto nos lleva a las tres secciones principales de un proyecto de Git: El directorio de Git (Git directory), el directorio de trabajo (working directory), y el área de preparación (staging area).

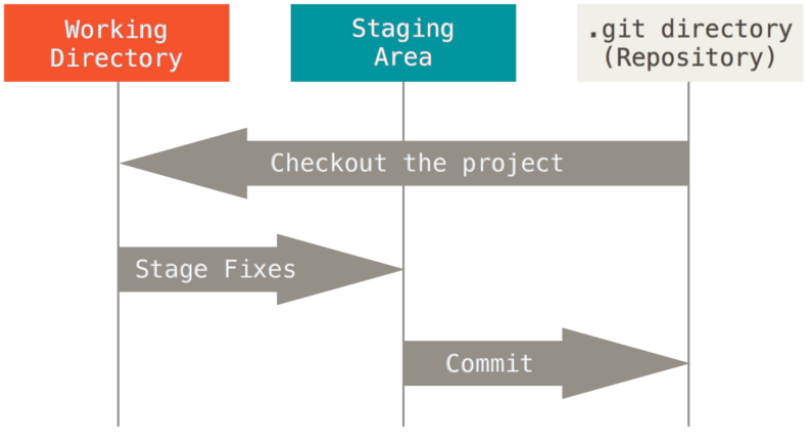


Figure 6. Directorio de trabajo, área de almacenamiento y el directorio Git.

El directorio de Git es donde se almacenan los metadatos y la base de datos de objetos para tu proyecto. Es la parte más importante de Git, y es lo que se copia cuando clonas un repositorio desde otra computadora.

El directorio de trabajo es una copia de una versión del proyecto. Estos archivos se sacan de la base de datos comprimida en el directorio de Git, y se colocan en disco para que los puedas usar o modificar.

El área de preparación es un archivo, generalmente contenido en tu directorio de Git, que almacena información acerca de lo que va a ir en tu próxima confirmación. A veces se le denomina índice (“index”), pero se está convirtiendo en estándar el referirse a ella como el área de preparación.

El flujo de trabajo básico en Git es algo así:

1. Modificas una serie de archivos en tu directorio de trabajo.
2. Preparas los archivos, añadiéndolos a tu área de preparación.
3. Confirmas los cambios, lo que toma los archivos tal y como están en el área de preparación y almacena esa copia instantánea de manera permanente en tu directorio de Git.

Si una versión concreta de un archivo está en el directorio de Git, se considera confirmada (committed). Si ha sufrido cambios desde que se obtuvo del repositorio, pero ha sido añadida al área de preparación, está preparada (staged). Y si ha sufrido cambios desde que se obtuvo del repositorio, pero no se ha preparado, está modificada (modified).

Configurando Git por primera vez

Ahora que tienes Git en tu sistema, vas a querer hacer algunas cosas para personalizar tu entorno de Git. Es necesario hacer estas cosas solamente una vez en tu computadora, y se mantendrán entre actualizaciones. También puedes cambiarlas en cualquier momento volviendo a ejecutar los comandos correspondientes.

Git trae una herramienta llamada git config, que te permite obtener y establecer variables de configuración que controlan el aspecto y funcionamiento de Git. Estas variables pueden almacenarse en tres sitios distintos:

1. Archivo /etc/gitconfig: Contiene valores para todos los usuarios del sistema y todos sus repositorios. Si pasas la opción --system a git config, lee y escribe específicamente en este archivo.
2. Archivo ~/.gitconfig o ~/.config/git/config: Este archivo es específico de tu usuario. Puedes hacer que Git lea y escriba específicamente en este archivo pasando la opción --global.
3. Archivo config en el directorio de Git (es decir, .git/config) del repositorio que estés utilizando actualmente: Este archivo es específico del repositorio actual.

Cada nivel sobrescribe los valores del nivel anterior, por lo que los valores de .git/config tienen preferencia sobre los de /etc/gitconfig.

En sistemas Windows, Git busca el archivo .gitconfig en el directorio $HOME (para mucha gente será (C:\Users\$USER). También busca el archivo /etc/gitconfig, aunque esta ruta es relativa a la raíz MSys, que es donde decidiste instalar Git en tu sistema Windows cuando ejecutaste el instalador.

Tu Identidad

Lo primero que deberás hacer cuando instales Git es establecer tu nombre de usuario y dirección de correo electrónico. Esto es importante porque los "commits" de Git usan esta información, y es introducida de manera inmutable en los commits que envías:

$ git config --global user.name "John Doe"

$ git config --global user.email johndoe@example.com

De nuevo, sólo necesitas hacer esto una vez si especificas la opción --global, ya que Git siempre usará esta información para todo lo que hagas en ese sistema. Si quieres sobrescribir esta información con otro nombre o dirección de correo para proyectos específicos, puedes ejecutar el comando sin la opción --global cuando estés en ese proyecto.

Muchas de las herramientas de interfaz gráfica te ayudarán a hacer esto la primera vez que las uses.

Comprobando tu Configuración

Si quieres comprobar tu configuración, puedes usar el comando git config --list para mostrar todas las propiedades que Git ha configurado:

$ git config --list

user.name=John Doe

user.email=johndoe@example.com

color.status=auto

color.branch=auto

color.interactive=auto

color.diff=auto

...

Puede que veas claves repetidas, porque Git lee la misma clave de distintos archivos (/etc/gitconfig y ~/.gitconfig, por ejemplo). En estos casos, Git usa el último valor para cada clave única que ve.

También puedes comprobar el valor que Git utilizará para una clave específica ejecutando **git config <key>:**

$ git config user.name

John Doe

Obteniendo un repositorio en Git

Puedes obtener un proyecto Git de dos maneras. La primera es tomar un proyecto o directorio existente e importarlo en Git. La segunda es clonar un repositorio existente en Git desde otro servidor.

Inicializando un repositorio en un directorio existente

Si estás empezando a seguir un proyecto existente en Git, debes ir al directorio del proyecto y usar el siguiente comando: **git init**

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

Clonando un repositorio existente

Si deseas obtener una copia de un repositorio Git existente — por ejemplo, un proyecto en el que te gustaría contribuir — el comando que necesitas es git clone. Si estás familizarizado con otros sistemas de control de versiones como Subversion, verás que el comando es "clone" en vez de "checkout". Es una distinción importante, ya que Git recibe una copia de casi todos los datos que tiene el servidor. Cada versión de cada archivo de la historia del proyecto es descargada por defecto cuando ejecutas git clone. De hecho, si el disco de tu servidor se corrompe, puedes usar cualquiera de los clones en cualquiera de los clientes para devolver el servidor al estado en el que estaba cuando fue clonado (puede que pierdas algunos hooks del lado del servidor y demás, pero toda la información acerca de las versiones estará ahí).

Puedes clonar un repositorio con git clone [url]. Por ejemplo, si quieres clonar la librería de Git llamada libgit2 puedes hacer algo así:

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

Esto crea un directorio llamado libgit2, inicializa un directorio .git en su interior, descarga toda la información de ese repositorio y saca una copia de trabajo de la última versión. Si te metes en el directorio libgit2, verás que están los archivos del proyecto listos para ser utilizados. Si quieres clonar el repositorio a un directorio con otro nombre que no sea libgit2, puedes especificarlo con la siguiente opción de línea de comandos:

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

Ese comando hace lo mismo que el anterior, pero el directorio de destino se llamará mylibgit.

Seguimiento básico en Git

¿Cómo le digo a Git que quiero que determinados archivos en este directorio sean “seguidos” para corroborar sus cambios?

Primero que nada, tipearemos por consola el siguiente comando:

git status

Podremos apreciar ahora en color rojo los archivos e incluso carpetas que no están siendo ‘seguidos’. Es el turno de la utilización de otro comando para agregar este tipo de archivos al **stage**, la cual es una zona que ss considerada como una zona intermediaria en donde los archivos de nuestro repositorio esperan ser confirmados.

Como mencionamos anteriormente el comando que nos ayudará a realizar todo esto previo (mandar a todos los archivos o los que nosotros deseemos al stage) es el de **git add**.

Existen distintas formas de utilizarlo:

- **git add .** -> Agrega en su totalidad archivos y carpetas existentes dentro del proyecto.
- **git add - - all** -> Agrega en su totalidad archivos y carpetas dentro del proyecto.
- **git add** -**A** -> Agrega en su totalidad archivos y carpetas dentro del proyecto.
- **git add nombreArchivo** -> Agrega ese archivo al stage.
- **git add nombreCarpeta/** -> Agrega todos los archivos pertenecientes a esa carpeta.
- **git add nombreCarpeta/*.extensión** -> Agrega los archivos de esa carpeta con esa extensión.

De momento estamos en el **stage**, es hora de que confirmemos estos archivos para que puedan formar parte como un hito en el tiempo, es hora de **commitear**.

¿Cómo los confirmamos? git commit -m "Nombre_commit", ahora podremos tener un control de esos archivos y si ingresamos **git status** no nos mostrará de momento algún archivo para agregar al stage. Dado que ya fueron commiteados.

Si yo modifico el contenido de alguno de los archivos que están en control ahora, y procedo a tipear en el bash **git status** veremos que se ha modificado el archivo donde realizamos los cambios y aparecerá en **rojo** puesto que no está siendo seguido.

¿Cómo se puede hacer para ver cuál/es ha/n sido esa/s modificación/es que han alterado el status de git? Con el comando **git diff** podremos hacer cosas muy interesantes.

Antes de haber hecho un 'commit': supongamos que hemos agregado 2 archivos al **stage** y luego los hemos modificado, simplemente con un **git diff** podremos apreciar cuáles son los cambios que hemos incorporado en ambos archivos (SIEMPRE Y CUANDO no los hayamos sumado al **stage)**

Después de haber hecho un 'commit': Generalmente esto es lo que suele pasar, luego de que hemos ya asentado a esos 2 archivos mencionados anteriormente y luego haber realizado ciertas modificaciones a los mismos del ejemplo anterior, podremos utilizar **git diff** para ver cómo era e l estado de esos 2 archivos en el commit anterior SIEMPRE Y CUANDO no los sumemos al **stage** (estos archivos modificados), una vez hecho esto el comando no funcionaría.

Historial de confirmaciones o de hitos

A veces necesitamos examinar la secuencia de commits (confirmaciones) que hemos realizado en la historia de un proyecto, para saber dónde estamos y qué estados hemos tenido a lo largo de la vida del repositorio. Esto lo conseguimos con el comando de Git:

git log

Con tipear este comando en el bash de **Git** podremos apreciar el histórico de commits, estando situados en la carpeta de nuestro proyecto.

Como puedes observar, el listado de commits está invertido, es decir, los últimos realizados aparecen los primeros.

De un commit podemos ver diversas informaciones básicas como:

- Identificador del commit

- Autor
- Fecha de realización
- Mensaje enviado

Log en una línea por commit

Es muy útil lanzar el log en una sola línea, lo que permite que veamos más cantidad de commits en la pantalla y facilita mucho seguir la secuencia, en vez de tener que ver un montón de páginas de commits.

Para ello usamos el mismo comando, pero con la opción "--oneline":

git log --oneline

Ver un número limitado de commits

Si tu proyecto ya tiene muchos commits, decenas o cientos de ellos, quizás no quieras mostrarlos todos, ya que generalmente no querrás ver cosas tan antiguas como el origen del repositorio. Para ver un número de logs determinado introducimos ese número como opción, con el signo "-" delante (-1, -8, -12...).

Por ejemplo, esto muestra los últimos tres commits:

git log -3

Ver información extendida del commit

Si queremos que el log también nos muestre los cambios en el código de cada commit podemos usar la opción -p. Esta opción genera una salida mucho más larga, por lo que seguramente nos tocará movernos en la salida con los cursores y usaremos CTRL + Z para salir.

git log -2 -p

Eso nos mostrará los cambios en el código de los últimos dos commits. Sin embargo, si quieres ver los cambios de cualquier otro commit que no sea tan reciente es mucho más cómodo usar otro comando de git que te explicamos a continuación "git show".

Comando "show": obtener mayor información de un commit

Podemos ver qué cambios en el código se produjeron mediante un commit en concreto con el comando "show". No es algo realmente relativo al propio log del commit, pero sí que necesitamos hacer log primero para ver el identificador del commit que queremos examinar.

Realmente nos vale con indicarle el resumen de identificador del commit, que vimos con el modificador --oneline.

git show b2c07b2

Podrás observar que al mostrar la información del commit te indica todas las líneas agregadas, en verde y con el signo "+" al principio , y las líneas quitadas en rojo y con el signo "-" al principio.

Opción --graph para el log con diagrama de las ramas

Si tu repositorio tiene ramas (branch) y quieres que el log te muestre más información de las ramas existentes, sus uniones (merges) y cosas así, puedes hacer uso de la opción --graph.

git log --graph --oneline

Ese comando te mostrará los commit en una línea y las ramas en las que estabas, con sus diferentes operaciones.

Fundamentos en Git para deshacer cosas

En cualquier momento puede que quieras deshacer algo. Aquí repasaremos algunas herramientas básicas usadas para deshacer cambios que hayas hecho. Ten cuidado, a veces no es posible recuperar algo luego que lo has deshecho. Esta es una de las pocas áreas en las que Git puede perder parte de tu trabajo si cometes un error.

Uno de las acciones más comunes a deshacer es cuando confirmas un cambio antes de tiempo y olvidas agregar algún archivo, o te equivocas en el mensaje de confirmación. Si quieres rehacer la confirmación, puedes reconfirmar con la opción --amend:

git commit –amend

Este comando utiliza tu área de preparación para la confirmación. Si no has hecho cambios desde tu última confirmación (por ejemplo, ejecutas este comando justo después de tu confirmación anterior), entonces la instantánea lucirá exactamente igual y lo único que cambiarás será el mensaje de confirmación.

Se lanzará el mismo editor de confirmación, pero verás que ya incluye el mensaje de tu confirmación anterior. Puedes editar el mensaje como siempre y se sobreescribirá tu confirmación anterior.

Por ejemplo, si confirmas y luego te das cuenta que olvidaste preparar los cambios de un archivo que querías incluir en esta confirmación, puedes hacer lo siguiente:

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

Al final terminarás con una sola confirmación - la segunda confirmación reemplaza el resultado de la primera.

Por ejemplo, algo común que suele pasar es el de que hemos commiteado y le hemos colocado un título con errores de ortografía o no hemos aclarado como nosotros quisimos a que hacía referencia ese determinado hito, con: **git commit --amend -m “ ”** para poder hacer una rectificación, cuando tipeemos **git log** y veamos el registro con nuestros commits podremos observar que el commit que no era deseado (en cuanto al título) ya no está y el que estará en su lugar es el nuevo commit rectificando al anterior.

¿Existe la posibilidad de dejar de realizar ese seguimiento de ese/esos mismo/s archivo/s que está/n dentro del stage? Claro que sí, **git** nos ofrece el comando **git reset**

El comando git reset se utiliza sobre todo para deshacer las cosas, como posiblemente puedes deducir por el verbo. Se mueve alrededor del puntero HEAD y opcionalmente cambia el index o área de ensayo y también puede cambiar opcionalmente el directorio de trabajo si se utiliza --hard. Esta última opción hace posible que este comando pueda perder tu trabajo si se usa incorrectamente, por lo que asegúrese de entenderlo antes de usarlo.

Existen distintas formas de utilizarlo:

- **git reset** -> Quita del stage todos los archivos y carpetas del proyecto.
- **git reset nombreArchivo** -> Quita del stage el archivo indicado.
- **git reset nombreCarpeta/** -> Quita del stage todos los archivos de esa carpeta.
- **git reset nombreCarpeta/nombreArchivo** -> Quita ese archivo del stage (que a la vez está dentro de una carpeta).
- **git reset nombreCarpeta/*.extensión** -> Quita todos los archivos que cumplan con la condición indicada previamente dentro de esa carpeta del stage.

¿Qué tal si te das cuenta que no quieres mantener los cambios que has realizado en determinado/s archivo/s? Git nos ofrece la posibilidad de restaurar ese/os archivo/s tal como estaban antes cuando estaba/n en el **stage**.

El comando que nos brinda es: **git checkout - - nombreArchivo o nombreCarpeta/nombreArchivo**

De esta manera procedemos a reestablecer el archivo en el estado en el que se encontraba previo a la modificación, esto obviamente siempre y cuando no hayamos hecho un **commit** para confirmar este mismo cambio.

¿Qué pasa si creamos un nuevo archivo, le colocamos su nombre y luego queremos cambiarle el mismo? Con el comando **git mv nombre_anterior nombre_nuevo** , lo bueno de esto es que si bien hemos modificado de manera superficial a nuestro archivo anterior cambiándole el nombre por uno nuevo, cuando tipeemos **git status** nos mostrará que se ha renombrado el archivo anterior por uno nuevo y que este último sigue estando en el stage y solamente restaría si lo deseamos, hacer el commit respectivo.

Otro escenario: nos arrepentimos de haber creado un archivo y lo queremos remover, podríamos eliminarlo manualmente y luego hacer el commit respectivo, pero deseamos llevar un control de todos los cambios que hacemos, entonces lo que vamos a hacer es tipear en el bash de Git el siguiente comando: **git rm nombreArchivo** (con su respectiva extensión) con **git status** podremos observar que el archivo ha sido eliminado.

Pero puede pasar que luego de un tiempo nos dimos cuenta que ese archivo era importante dado que el código que estaba en él era útil y queremos volverlo a utilizar **¿cómo podré recuperar ese**

archivo si ya lo borre (sabiendo de que ese mismo archivo no esta en la papelera de reciclaje)? Git puede hacer algo por nosotros con el comando **git reset --soft idCommit** para hacer un paso atrás en el tiempo, tipeamos **git log** y veremos que el commit donde se mencionaba que se había borrado determinado archivo ya no existe

Pero si colocamos **git status** veremos que se verá en el stage que hemos eliminado ese archivo, dado que **git reset --soft idCommit** no nos recupera el archivo en si, solamente se sitúa en ese lugar y nos podría mostrar si es que hubo alguna modificación en el archivo lo podrías ver. Si queremos recuperar de manera virtual el archivo borrado deberemos de tipear el siguiente comando: **git reset - - hard idCommit** (obviamente la id del commit anterior al borrado del archivo) de esta manera no solo recuperamos el archivo, sino que cuando hagamos un **git status** no se verán cambios y con **git log** no veremos el commit que mostraba que se había borrado un archivo.

¿Cómo nos damos cuenta que en un momento se borró algún archivo? Con **git reflog** podremos observar este tipo de cambio y otros más, es más es ideal para lograr identificar las id's de los commits.

git reset - - mixed idCommit : va a un punto de la historia, es decir, un punto de nuestros hitos evitando los commits que se hicieron desde ese punto en adelante, pero los cambios se van a mantener. Lo que hago en si es que la id que digité en el comando ayudará a identificarlo y quitar del stage sus respectivos cambios.

Conclusión con respecto a **soft**, **hard** y **mixed** :

Soft

- "elimina" los commits posteriores al commit al que estas haciendo el reset
- conserva los cambios en el stage area
- conserva los cambios que tengas en tus archivos (working directory)

Mixed

- "elimina" los commits posteriores al commit al que estas haciendo el reset
- Deshace los cambios en el stage area
- conserva los cambios que tengas en tus archivos (working directory)

Hard

- "elimina" los commits posteriores al commit al que estás haciendo el reset
- Deshace los cambios en el stage area
- Deshace los cambios que tengas en tus archivos (working directory)

GITIGNORE

No siempre es bueno llevar un control de todos los archivos de nuestro proyecto, muchas veces tendremos una buena cantidad de archivos que no tendrán mucha relación con el proyecto en si, muchos de ellos generados cuando trabajamos con ide's. Por lo tanto es mejor ignorarlos, ahora se mostrará cómo hacerlo.

Deberemos primero que nada crear un nuevo archivo '**.gitignore'** (el cual debe de estar a la altura de la carpeta **.git)** . En el indicaremos el nombre del archivo que no queremos que se le siga un control, es decir, que no queremos mandarlo al stage. Cuando tipeemos el **git status** ya no aparecerá ese archivo no deseado, y si aparecerá en rojo **.gitignore** , debemos de agregarlo al stage a este último.

¿Cómo decirle a .gitignore que no lleve el seguimiento de varios archivos o de unos con extensión determinada?

Por ejemplo, tenemos 2 archivos: ide-config.html y ide-config2.html , estos mismos no los queremos, por lo tanto lo recomendado es hacer lo siguiente, en el archivo **.gitignore** tipear

*ide-config

O

Ide-config*

Lo recomendable es tener en una carpeta cada uno de esos archivos indeseados, para tener un orden y sea más fácil poder ignorarlos desde **.gitignore.** Por ejemplo creamos una carpeta llamada "config" en ella situamos a esos 2 archivos que creamos con anterioridad y en nuestro archivo ignorador, colocamos **/config/** guardamos, vamos a **git status** veremos que esos archivos no están habilitados para agregarlos al stage y habrá funcionado correctamente.

Es recomendable visitar páginas como **github** donde los usuarios de esta comunidad han creado repositorios con líneas de código recomendadas para poder colocarlas en el **.gitignore** y evitar que esos archivos, como bien mencionamos anteriormente, generados por los ide's sean ignorados.

https://www.github.com/github/gitignore

Alias de Git

Git no deduce automáticamente tu comando si lo tecleas parcialmente. Si no quieres teclear el nombre completo de cada comando de Git, puedes establecer fácilmente un alias para cada comando mediante git config. Aquí tienes algunos ejemplos que te pueden interesar:

$ git config --global alias.co checkout

$ git config --global alias.br branch

$ git config --global alias.ci commit

$ git config --global alias.st status

Esto significa que, por ejemplo, en lugar de teclear git commit, solo necesitas teclear git ci. A medida que uses Git, probablemente también utilizarás otros comandos con frecuencia; no dudes en crear nuevos alias para ellos.

Esta técnica también puede resultar útil para crear comandos que en tu opinión deberían existir. Por ejemplo, para corregir el problema de usabilidad que encontraste al quitar del área de preparación un archivo, puedes añadir tu propio alias a Git:

$ git config --global alias.unstage 'reset HEAD --'

Esto hace que los dos comandos siguientes sean equivalentes:

$ git unstage fileA

$ git reset HEAD fileA

Esto parece un poco más claro. También es frecuente añadir un comando last, de este modo:

$ git config --global alias.last 'log -1 HEAD'

De esta manera, puedes ver fácilmente cuál fue la última confirmación:

$ git last

commit 66938dae3329c7aebe598c2246a8e6af90d04646

Author: Josh Goebel <dreamer3@example.com>

Date: Tue Aug 26 19:48:51 2008 +0800

test for current head

Signed-off-by: Scott Chacon <schacon@example.com>

Como puedes ver, Git simplemente sustituye el nuevo comando por lo que sea que hayas puesto en el alias. Sin embargo, quizás quieras ejecutar un comando externo en lugar de un subcomando de Git. En ese caso, puedes comenzar el comando con un carácter !. Esto resulta útil si escribes tus propias herramientas para trabajar con un repositorio de Git. Podemos demostrarlo creando el alias git visual para ejecutar gitk:

$ git config --global alias.visual "!gitk"

Etiquetado

Como muchos VCS, Git tiene la posibilidad de etiquetar puntos específicos del historial como importantes. Esta funcionalidad se usa típicamente para marcar versiones de lanzamiento (v1.0, por ejemplo). En esta sección, aprenderás cómo listar las etiquetas disponibles, cómo crear nuevas etiquetas y cuáles son los distintos tipos de etiquetas.

Listar Tus Etiquetas

Listar las etiquetas disponibles en Git es sencillo. Simplemente escribe git tag:

$ git tag

v0.1

v1.3

Este comando lista las etiquetas en orden alfabético; el orden en el que aparecen no tiene mayor importancia.

También puedes buscar etiquetas con un patrón particular. El repositorio del código fuente de Git, por ejemplo, contiene más de 500 etiquetas. Si sólo te interesa ver la serie 1.8.5, puedes ejecutar:

$ git tag -l 'v1.8.5*'

v1.8.5

v1.8.5-rc0

v1.8.5-rc1

v1.8.5-rc2

v1.8.5-rc3

v1.8.5.1

v1.8.5.2

v1.8.5.3

v1.8.5.4

v1.8.5.5

Crear Etiquetas

Git utiliza dos tipos principales de etiquetas: ligeras y anotadas.

Una etiqueta ligera es muy parecida a una rama que no cambia - simplemente es un puntero a un **commit** específico.

Sin embargo, las etiquetas anotadas se guardan en la base de datos de Git como objetos enteros. Tienen un **checksum**; contienen el nombre del etiquetador, correo electrónico y fecha; tienen un mensaje asociado; y pueden ser firmadas y verificadas con **GNU Privacy Guard** (GPG). Normalmente se recomienda que crees etiquetas anotadas, de manera que tengas toda esta información; pero si quieres una etiqueta temporal o por alguna razón no estás interesado en esa información, entonces puedes usar las etiquetas ligeras.

Etiquetas Anotadas

Crear una etiqueta anotada en Git es sencillo. La forma más fácil de hacerlo es especificar la opción -a cuando ejecutas el comando git tag:

$ git tag -a v1.4 -m 'my version 1.4'

$ git tag

v0.1

v1.3

v1.4

La opción -m especifica el mensaje de la etiqueta, el cual es guardado junto con ella. Si no especificas el mensaje de una etiqueta anotada, Git abrirá el editor de texto para que lo escribas.

Puedes ver la información de la etiqueta junto con el **commit** que está etiquetado al usar el comando git show:

$ git show v1.4

tag v1.4

Tagger: Ben Straub <ben@straub.cc>

Date: Sat May 3 20:19:12 2014 -0700

my version 1.4

commit ca82a6dff817ec66f44342007202690a93763949

Author: Scott Chacon <schacon@gee-mail.com>

Date: Mon Mar 17 21:52:11 2008 -0700

changed the version number

El comando muestra la información del etiquetador, la fecha en la que el **commit** fue etiquetado y el mensaje de la etiqueta, antes de mostrar la información del **commit**.

Etiquetas Ligeras

La otra forma de etiquetar un **commit** es mediante una etiqueta ligera. Una etiqueta ligera no es más que el **checksum** de un **commit** guardado en un archivo - no incluye más información. Para crear una etiqueta ligera, no pases las opciones -a, -s ni -m:

$ git tag v1.4-lw

$ git tag

v0.1

v1.3

v1.4

v1.4-lw

v1.5

Esta vez, si ejecutas git show sobre la etiqueta no verás la información adicional. El comando solo mostrará el **commit**:

$ git show v1.4-lw

commit ca82a6dff817ec66f44342007202690a93763949

Author: Scott Chacon <schacon@gee-mail.com>

Date: Mon Mar 17 21:52:11 2008 -0700

changed the version number

Etiquetado Tardío

También puedes etiquetar **commits** mucho tiempo después de haberlos hecho. Supongamos que tu historial luce como el siguiente:

$ git log --pretty=oneline

15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'

a6b4c97498bd301d84096da251c98a07c7723e65 beginning write support

0d52aaab4479697da7686c15f77a3d64d9165190 one more thing

6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'

0b7434d86859cc7b8c3d5e1dddfed66ff742fcbc added a commit function

4682c3261057305bdd616e23b64b0857d832627b added a todo file

166ae0c4d3f420721acbb115cc33848dfcc2121a started write support

9fceb02d0ae598e95dc970b74767f19372d61af8 updated rakefile

964f16d36dfccde844893cac5b347e7b3d44abbc commit the todo

8a5cbc430f1a9c3d00faaeffd07798508422908a updated readme

Ahora, supongamos que olvidaste etiquetar el proyecto en su versión v1.2, la cual corresponde al **commit** “updated rakefile”. Igual puedes etiquetarlo. Para etiquetar un commit, debes especificar el **checksum** del **commit** (o parte de él) al final del comando:

$ git tag -a v1.2 9fceb02

Puedes ver que has etiquetado el commit:

$ git tag

v0.1

v1.2

v1.3

v1.4

v1.4-lw

v1.5

$ git show v1.2

tag v1.2

Tagger: Scott Chacon <schacon@gee-mail.com>

Date: Mon Feb 9 15:32:16 2009 -0800

version 1.2

commit 9fceb02d0ae598e95dc970b74767f19372d61af8

Author: Magnus Chacon <mchacon@gee-mail.com>

Date: Sun Apr 27 20:43:35 2008 -0700

updated rakefile

...

Sacar una Etiqueta

Si queremos eliminar una etiqueta con tipear **git tag -d nombreEtiqueta** y luego tipeemos el **git log** detallado, no lo veremos más.

Trabajar con Remotos

Para poder colaborar en cualquier proyecto Git, necesitas saber cómo gestionar repositorios remotos. Los repositorios remotos son versiones de tu proyecto que están hospedadas en Internet o en cualquier otra red. Puedes tener varios de ellos, y en cada uno tendrás generalmente permisos de solo lectura o de lectura y escritura. Colaborar con otras personas implica gestionar estos repositorios remotos enviando y trayendo datos de ellos cada vez que necesites compartir tu trabajo. Gestionar repositorios remotos incluye saber cómo añadir un repositorio remoto, eliminar los remotos que ya no son válidos, gestionar varias ramas remotas, definir si deben rastrearse o no y más. En esta sección, trataremos algunas de estas habilidades de gestión de remotos.

Ver Tus Remotos

Para ver los remotos que tienes configurados, debes ejecutar el comando **git remote**. Mostrará los nombres de cada uno de los remotos que tienes especificados. Si has clonado tu repositorio, deberías ver al menos **origin** (origen, en inglés) - este es el nombre que por defecto Git le da al servidor del que has clonado:

```
$ git clone https://github.com/schacon/ticgit
Cloning into 'ticgit'...
remote: Reusing existing pack: 1857, done.
remote: Total 1857 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (1857/1857), 374.35 KiB | 268.00 KiB/s, done.
Resolving deltas: 100% (772/772), done.
Checking connectivity... done.
$ cd ticgit
$ git remote
origin
```

También puedes pasar la opción -v, la cual muestra las URLs que Git ha asociado al nombre y que serán usadas al leer y escribir en ese remoto:

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

Si tienes más de un remoto, el comando los listará todos. Por ejemplo, un repositorio con múltiples remotos para trabajar con distintos colaboradores podría verse de la siguiente manera.

```
$ cd grit
$ git remote -v
bakkdoor  https://github.com/bakkdoor/grit (fetch)
bakkdoor  https://github.com/bakkdoor/grit (push)
cho45     https://github.com/cho45/grit (fetch)
cho45     https://github.com/cho45/grit (push)
defunkt   https://github.com/defunkt/grit (fetch)
defunkt   https://github.com/defunkt/grit (push)
koke      git://github.com/koke/grit.git (fetch)
koke      git://github.com/koke/grit.git (push)
origin    git@github.com:mojombo/grit.git (fetch)
origin    git@github.com:mojombo/grit.git (push)
```

Esto significa que podemos traer contribuciones de cualquiera de estos usuarios fácilmente. Es posible que también tengamos permisos para enviar datos a algunos, aunque no podemos saberlo desde aquí.

Añadir Repositorios Remotos

En secciones anteriores hemos mencionado y dado alguna demostración de cómo añadir repositorios remotos. Ahora veremos explícitamente cómo hacerlo. Para añadir un remoto nuevo

y asociarlo a un nombre que puedas referenciar fácilmente, ejecuta **git remote add [nombre] [url]:**

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

A partir de ahora puedes usar el nombre pb en la línea de comandos en lugar de la URL entera. Por ejemplo, si quieres traer toda la información que tiene Paul pero tú aún no tienes en tu repositorio, puedes ejecutar git fetch pb:

```
$ git fetch pb
remote: Counting objects: 43, done.
remote: Compressing objects: 100% (36/36), done.
remote: Total 43 (delta 10), reused 31 (delta 5)
Unpacking objects: 100% (43/43), done.
From https://github.com/paulboone/ticgit
 * [new branch]      master     -> pb/master
 * [new branch]      ticgit     -> pb/ticgit
```

La rama maestra de Paul ahora es accesible localmente con el nombre pb/master - puedes combinarla con alguna de tus ramas, o puedes crear una rama local en ese punto si quieres inspeccionarla.

Traer y Combinar Remotos

Como hemos visto hasta ahora, para obtener datos de tus proyectos remotos puedes ejecutar:

$ git fetch [remote-name]

El comando irá al proyecto remoto y se traerá todos los datos que aun no tienes de dicho remoto. Luego de hacer esto, tendrás referencias a todas las ramas del remoto, las cuales puedes combinar e inspeccionar cuando quieras.

Si clonas un repositorio, el comando de clonar automáticamente añade ese repositorio remoto con el nombre “origin”. Por lo tanto, git fetch origin se trae todo el trabajo nuevo que ha sido enviado a ese servidor desde que lo clonaste (o desde la última vez que trajiste datos). Es importante destacar que el comando git fetch solo trae datos a tu repositorio local - ni lo combina automáticamente con tu trabajo ni modifica el trabajo que llevas hecho. La combinación con tu trabajo debes hacerla manualmente cuando estés listo.

Si has configurado una rama para que rastree una rama remota (más información en la siguiente sección y en Ramificaciones en Git), puedes usar el comando git pull para traer y combinar automáticamente la rama remota con tu rama actual. Es posible que este sea un flujo de trabajo

mucho más cómodo y fácil para ti; y por defecto, el comando git clone le indica automáticamente a tu rama maestra local que rastree la rama maestra remota (o como se llame la rama por defecto) del servidor del que has clonado. Generalmente, al ejecutar git pull traerás datos del servidor del que clonaste originalmente y se intentará combinar automáticamente la información con el código en el que estás trabajando.

Enviar a Tus Remotos

Cuando tienes un proyecto que quieres compartir, debes enviarlo a un servidor. El comando para hacerlo es simple: git push [nombre-remoto] [nombre-rama]. Si quieres enviar tu rama master a tu servidor origin (recuerda, clonar un repositorio establece esos nombres automáticamente), entonces puedes ejecutar el siguiente comando y se enviarán todos los **commits** que hayas hecho al servidor:

$ git push origin master

Este comando solo funciona si clonaste de un servidor sobre el que tienes permisos de escritura y si nadie más ha enviado datos por el medio. Si alguien más clona el mismo repositorio que tú y envía información antes que tú, tu envío será rechazado. Tendrás que traerte su trabajo y combinarlo con el tuyo antes de que puedas enviar datos al servidor.

Inspeccionar un Remoto

Si quieres ver más información acerca de un remoto en particular, puedes ejecutar el comando git remote show [nombre-remoto]. Si ejecutas el comando con un nombre en particular, como origin, verás algo como lo siguiente:

```
$ git remote show origin
* remote origin
  Fetch URL: https://github.com/schacon/ticgit
  Push  URL: https://github.com/schacon/ticgit
  HEAD branch: master
  Remote branches:
    master                               tracked
    dev-branch                           tracked
  Local branch configured for 'git pull':
    master merges with remote master
  Local ref configured for 'git push':
    master pushes to master (up to date)
```

El comando lista la URL del repositorio remoto y la información del rastreo de ramas. El comando te indica claramente que, si estás en la rama maestra y ejecutas el comando git pull, automáticamente combinará la rama maestra remota con tu rama local, luego de haber traído toda la información de ella. También lista todas las referencias remotas de las que ha traído datos.

Ejemplos como este son los que te encontrarás normalmente. Sin embargo, si usas Git de forma más avanzada, puede que obtengas mucha más información de un git remote show:

```
$ git remote show origin
* remote origin
  URL: https://github.com/my-org/complex-project
  Fetch URL: https://github.com/my-org/complex-project
  Push  URL: https://github.com/my-org/complex-project
  HEAD branch: master
  Remote branches:
    master                           tracked
    dev-branch                       tracked
    markdown-strip                   tracked
    issue-43                         new (next fetch will store in remotes/origin)
    issue-45                         new (next fetch will store in remotes/origin)
    refs/remotes/origin/issue-11     stale (use 'git remote prune' to remove)
  Local branches configured for 'git pull':
    dev-branch merges with remote dev-branch
    master     merges with remote master
  Local refs configured for 'git push':
    dev-branch                     pushes to dev-branch                     (up to date)
    markdown-strip                 pushes to markdown-strip                 (up to date)
    master                         pushes to master                         (up to date)
```

Este comando te indica a cuál rama enviarás información automáticamente cada vez que ejecutas git push, dependiendo de la rama en la que estés. También te muestra cuáles ramas remotas no tienes aún, cuáles ramas remotas tienes que han sido eliminadas del servidor, y varias ramas que serán combinadas automáticamente cuando ejecutes git pull.

Eliminar y Renombrar Remotos

Si quieres cambiar el nombre de la referencia de un remoto puedes ejecutar git remote rename. Por ejemplo, si quieres cambiar el nombre de pb a paul, puedes hacerlo con **git remote rename**:

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

Es importante destacar que al hacer esto también cambias el nombre de las ramas remotas. Por lo tanto, lo que antes estaba referenciado como pb/master ahora lo está como paul/master.

Si por alguna razón quieres eliminar un remoto - has cambiado de servidor o no quieres seguir utilizando un **mirror** o quizás un colaborador ha dejado de trabajar en el proyecto - puedes usar git remote rm:

```
$ git remote rm paul
$ git remote
origin
```

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 tremendamente 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 padre 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 subdirectorio (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.

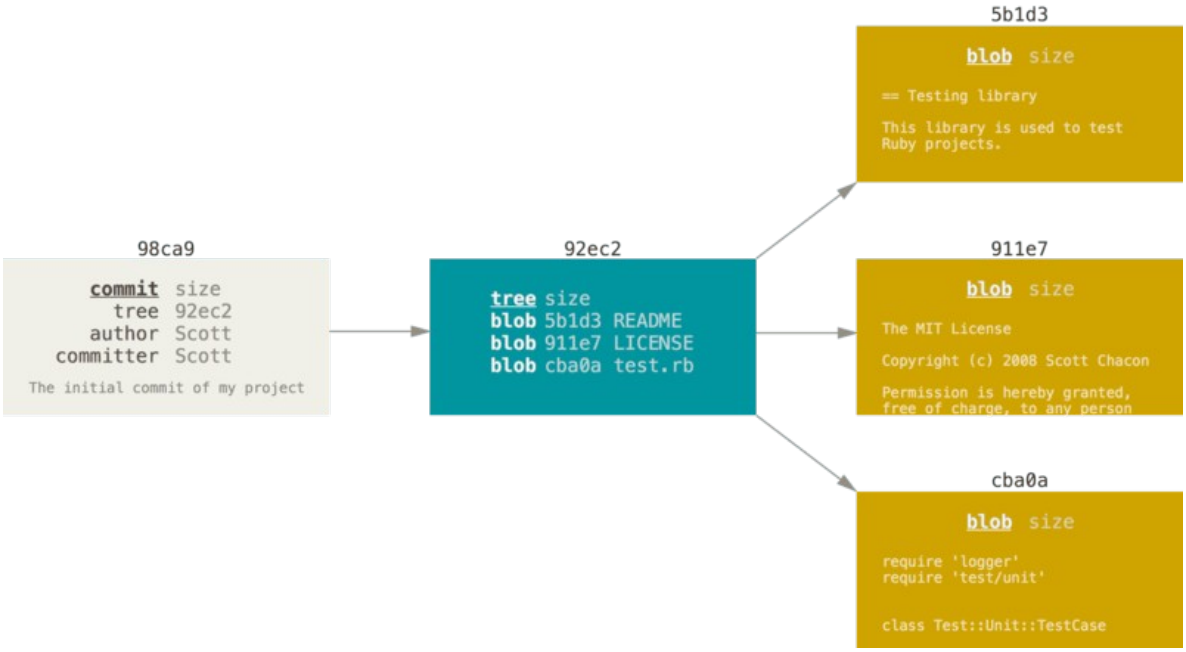


Figure 9. Una confirmación y sus árboles

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

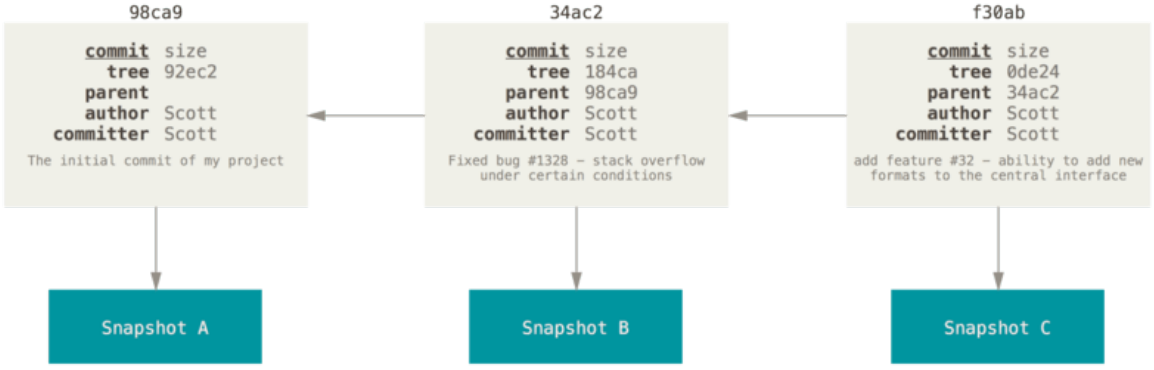


Figure 10. Confirmaciones y sus predecesoras

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.

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.

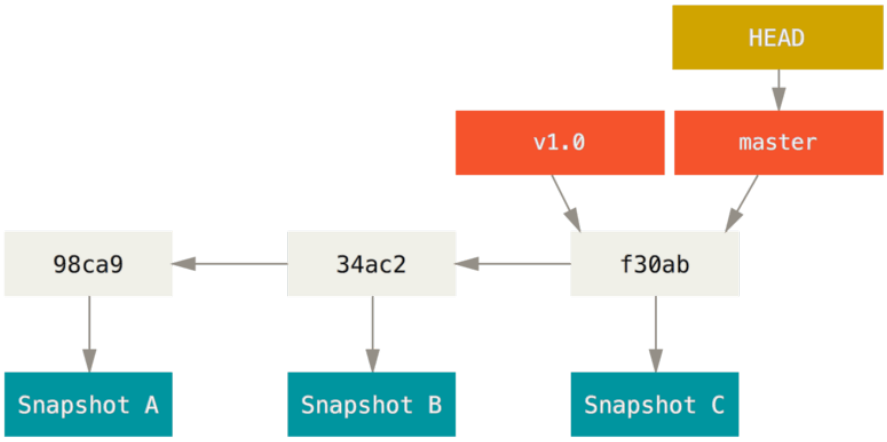


Figure 11. Una rama y su historial de confirmaciones

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.

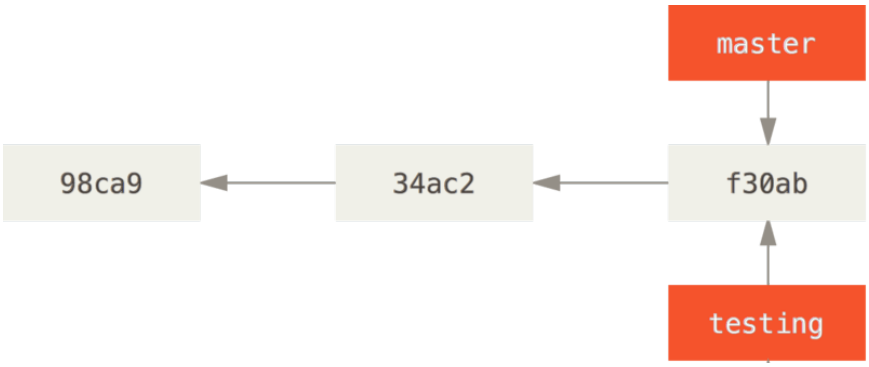


Figure 12. Dos ramas apuntando al mismo grupo de confirmaciones

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, pero no salta a dicha rama.

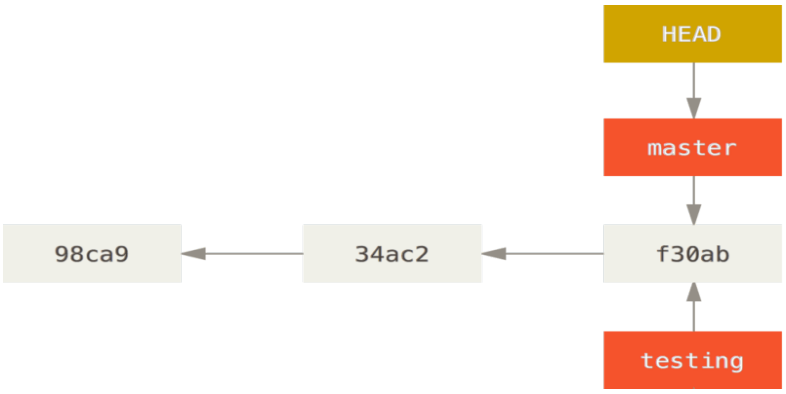


Figure 13. Apuntador HEAD a la rama donde estás actualmente

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.

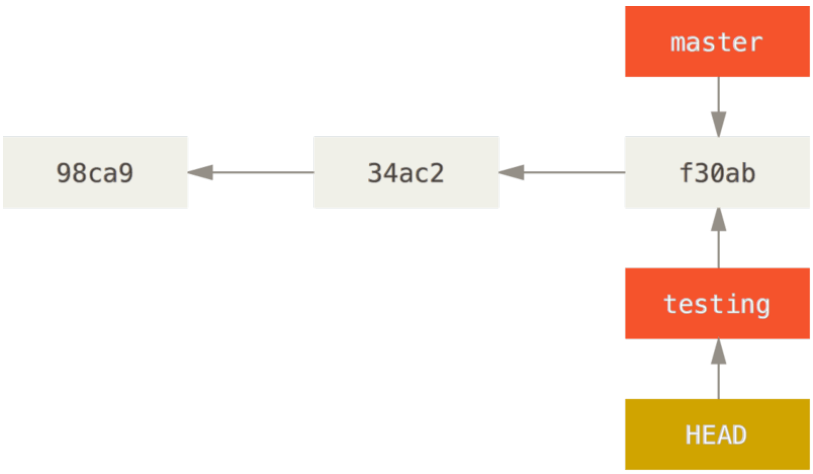


Figure 14. El apuntador HEAD apunta a la rama actual

¿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'

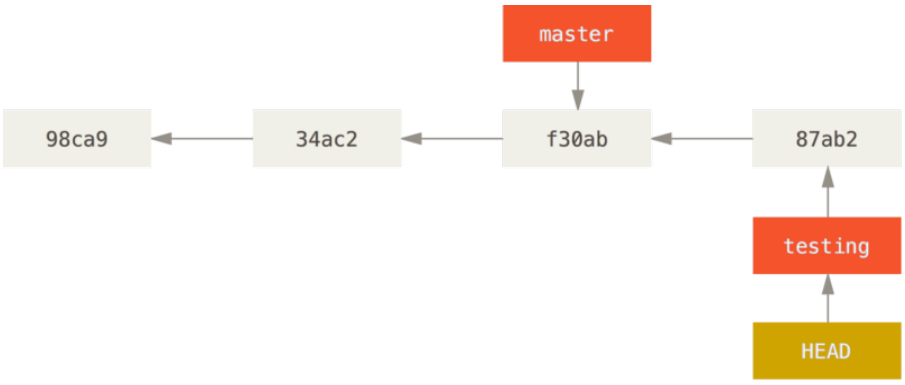


Figure 15. La rama apuntada por HEAD avanza con cada confirmación de cambios

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

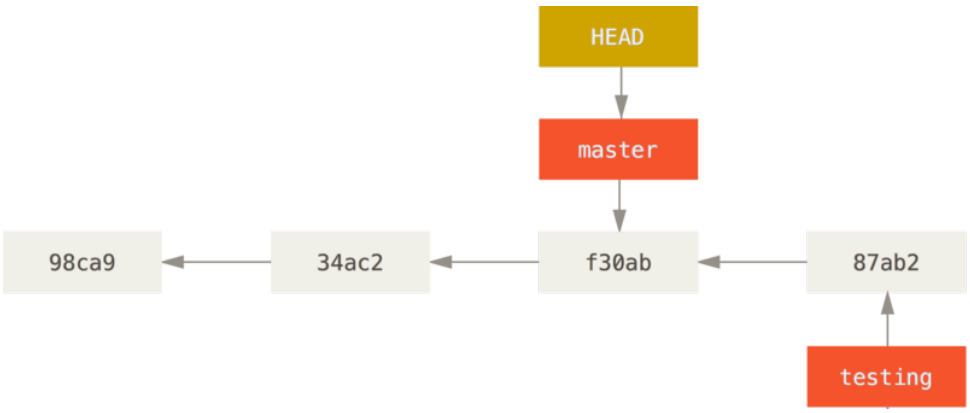


Figure 16. HEAD apunta a otra rama cuando hacemos un salto

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

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. 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.

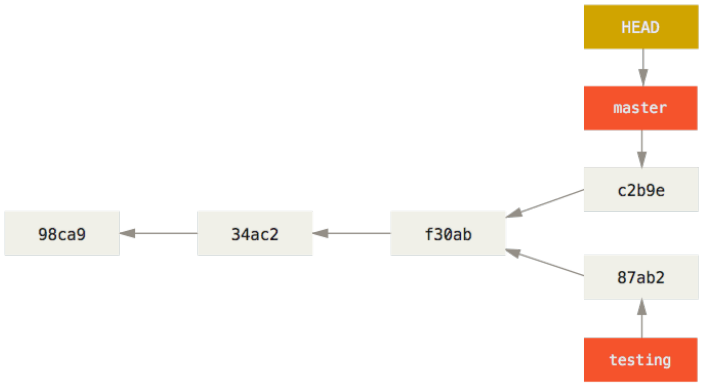


Figure 17. Los registros de las ramas divergen

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 continúas tu trabajo.

Procedimientos Básicos de Ramificación

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

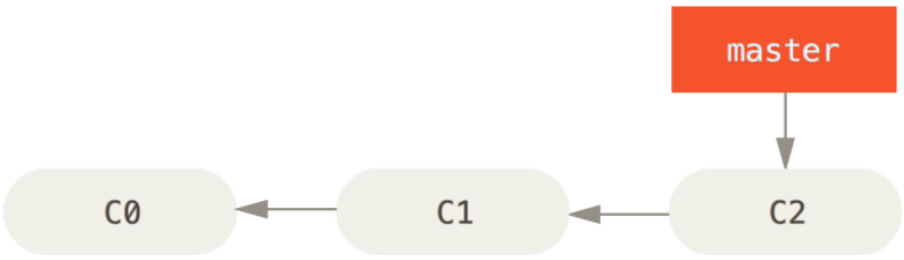


Figure 18. Un registro de confirmaciones corto y sencillo

Decides trabajar en el problema #53, según el sistema que tu compañía utiliza para llevar el 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 para:

$ git branch iss53

$ git checkout iss53

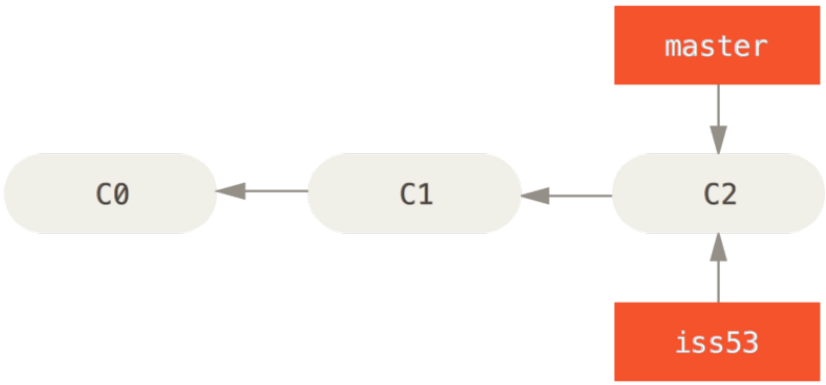


Figure 19. Crear un apuntador a la rama nueva

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]'

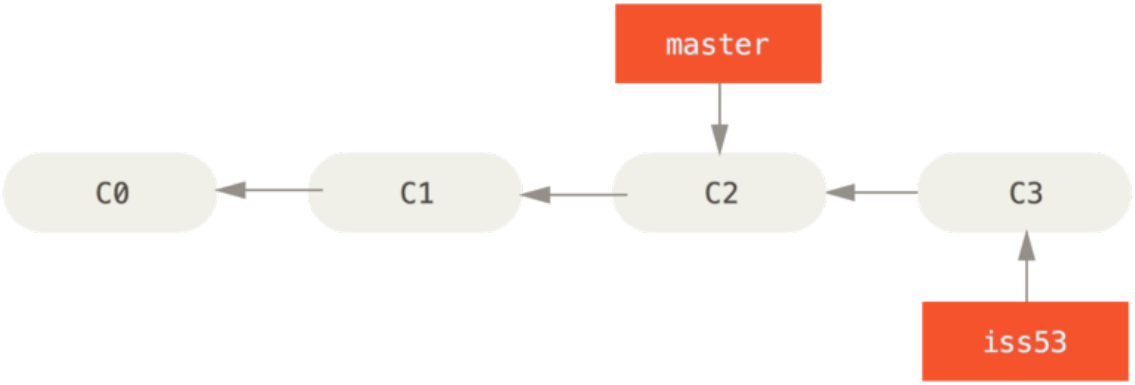


Figure 20. La rama iss53 ha avanzado con tu trabajo

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 tomar 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(+)
```

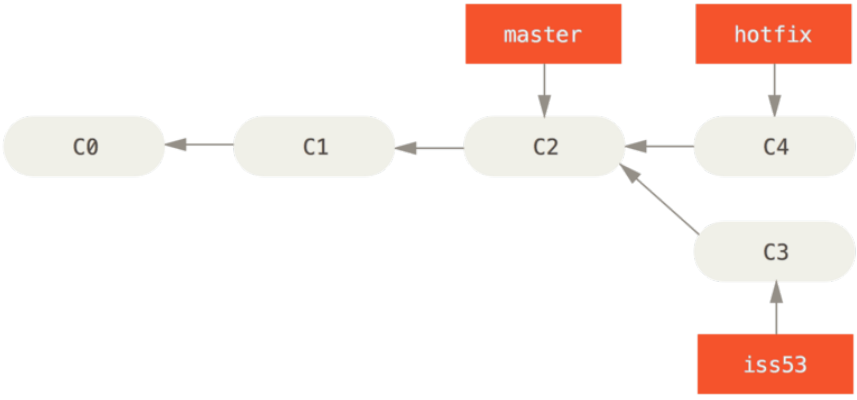


Figure 21. Rama hotfix basada en la rama master original

Puedes realizar las pruebas oportunas, asegurarte de 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.

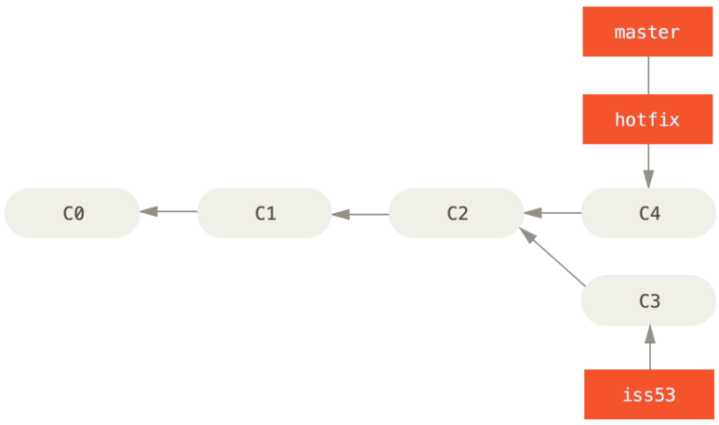


Figure 22. Tras la fusión (merge), la rama master apunta al mismo sitio que la rama hotfix.

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(+)
```

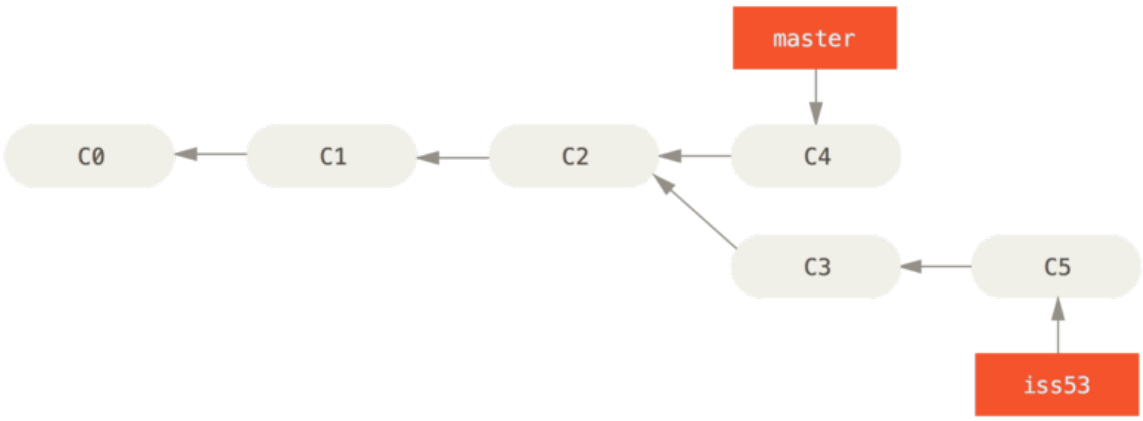


Figure 23. La rama iss53 puede avanzar independientemente

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.

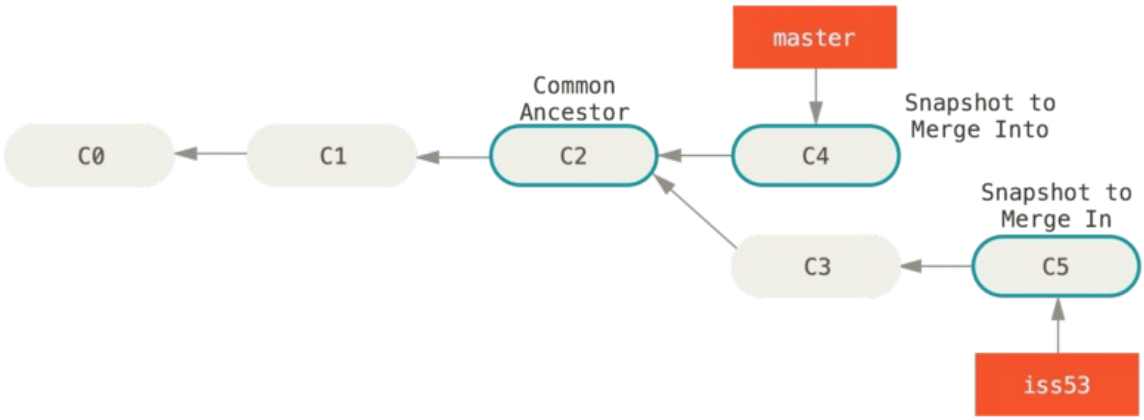


Figure 24. Git identifica automáticamente el mejor ancestro común para realizar la fusión de las ramas

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 padre.

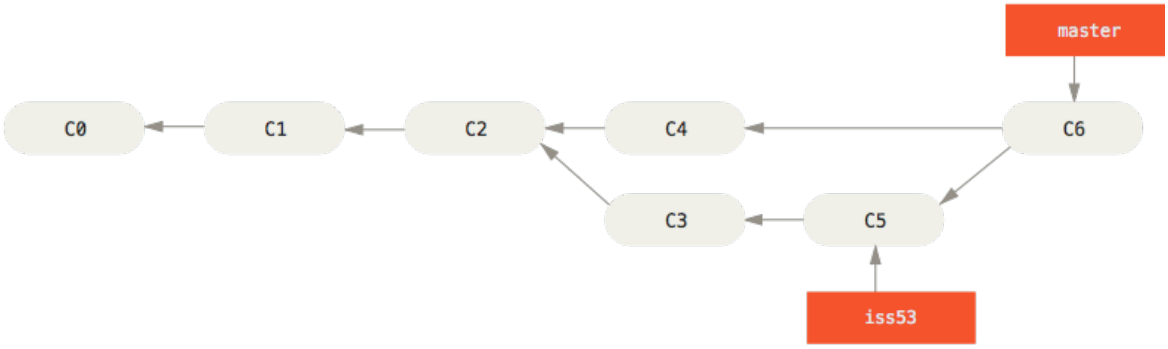


Figure 25. Git crea automáticamente una nueva confirmación para la fusión

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.

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.

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 con 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 intentar 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'.
```

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 irlas 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 suelen 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.

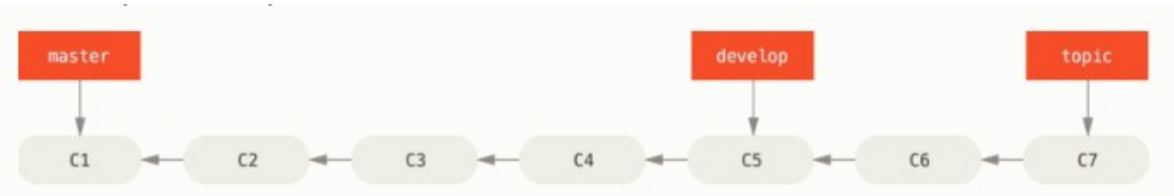


Figure 26. Una vista lineal del ramificado progresivo estable

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.

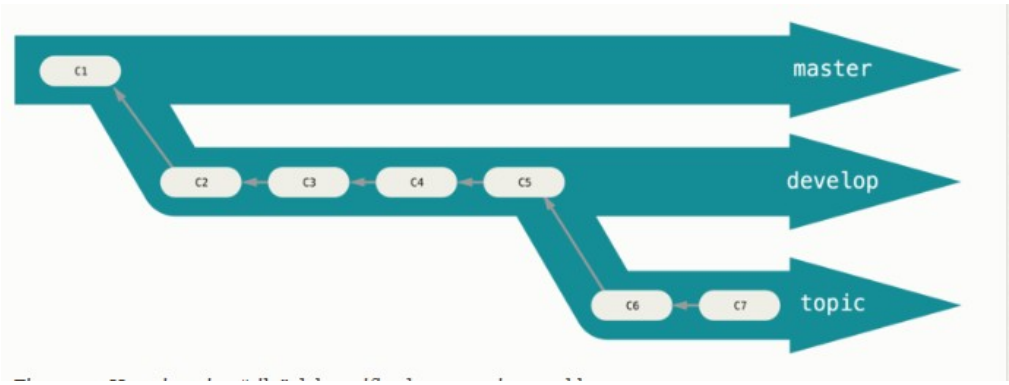


Figure 27. Una vista tipo “silo” del ramificado progresivo estable

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 que es 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 abres 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:

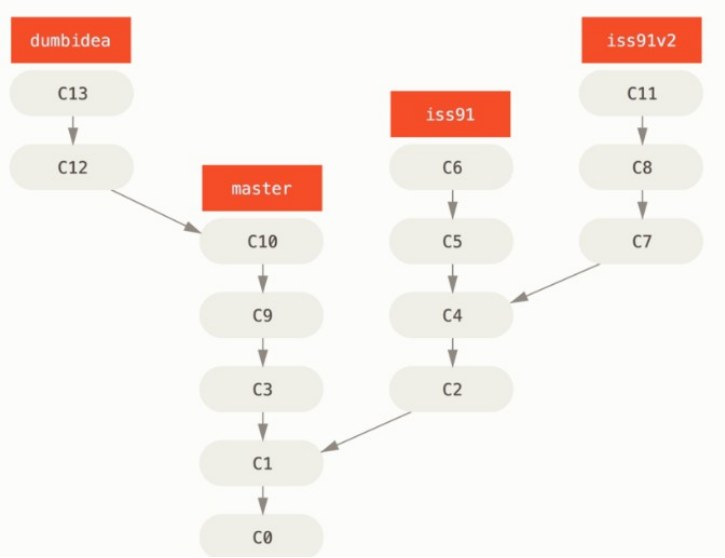
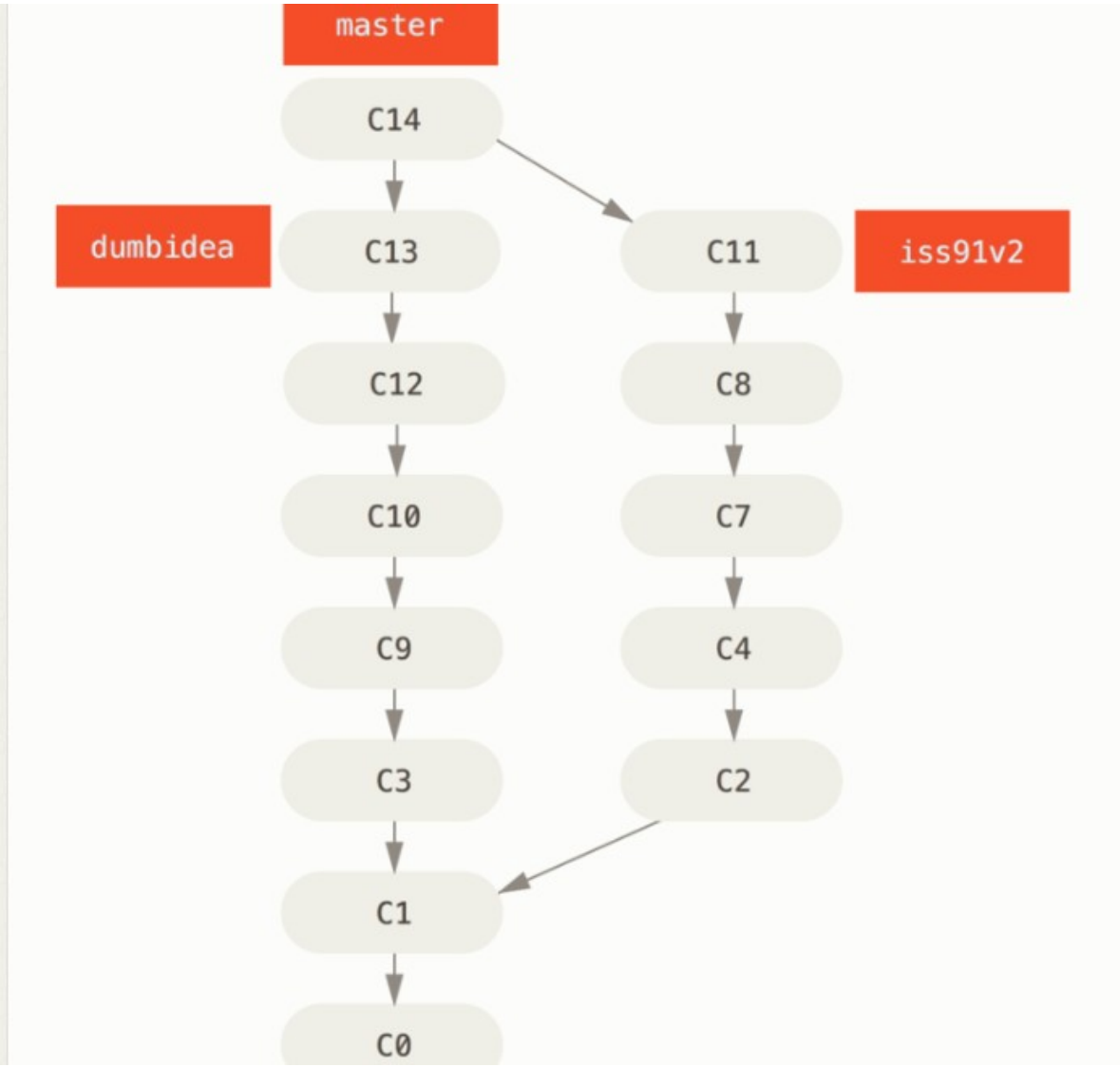


Figure 28. Múltiples ramas puntuales

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:



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.

Note

"origin" no es especial

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.

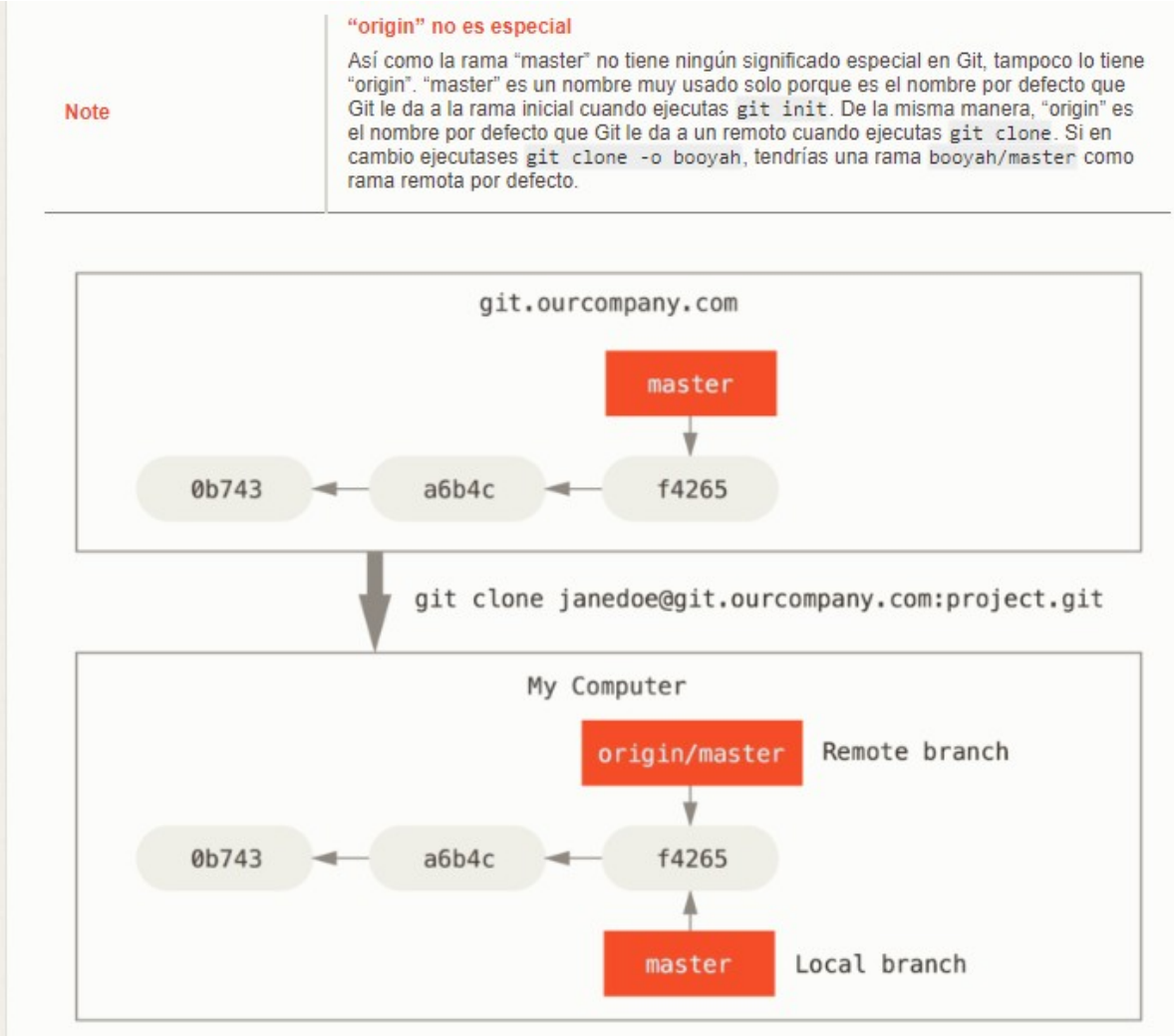


Figure 30. Servidor y repositorio local luego de ser clonado

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.

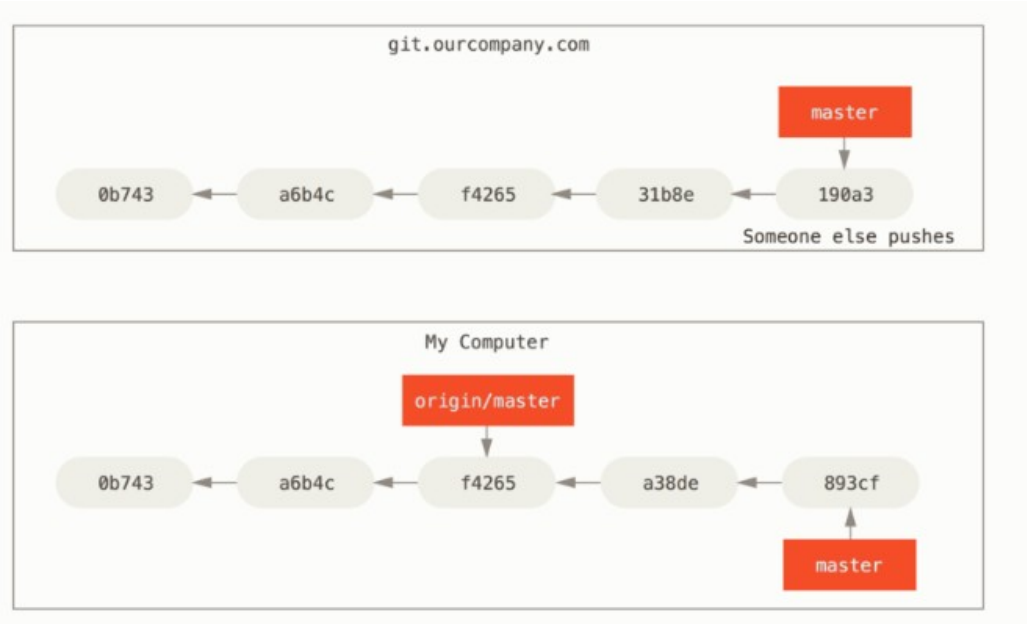


Figure 31. El trabajo remoto y el local pueden diverger

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.

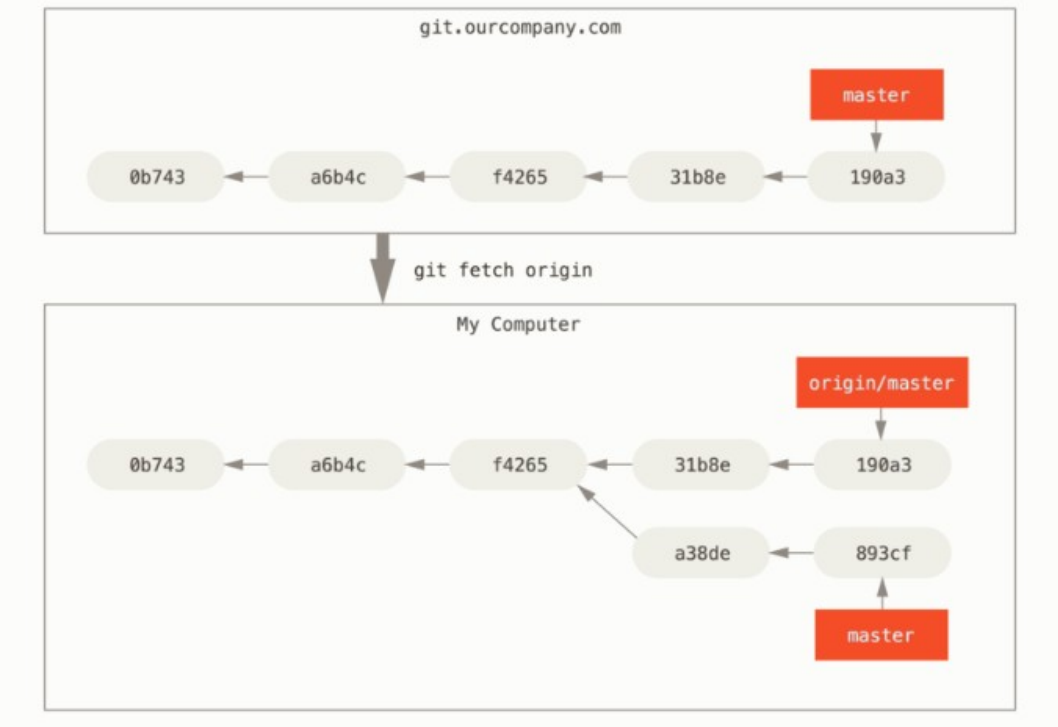


Figure 32. git fetch actualiza las referencias de tu remoto

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.

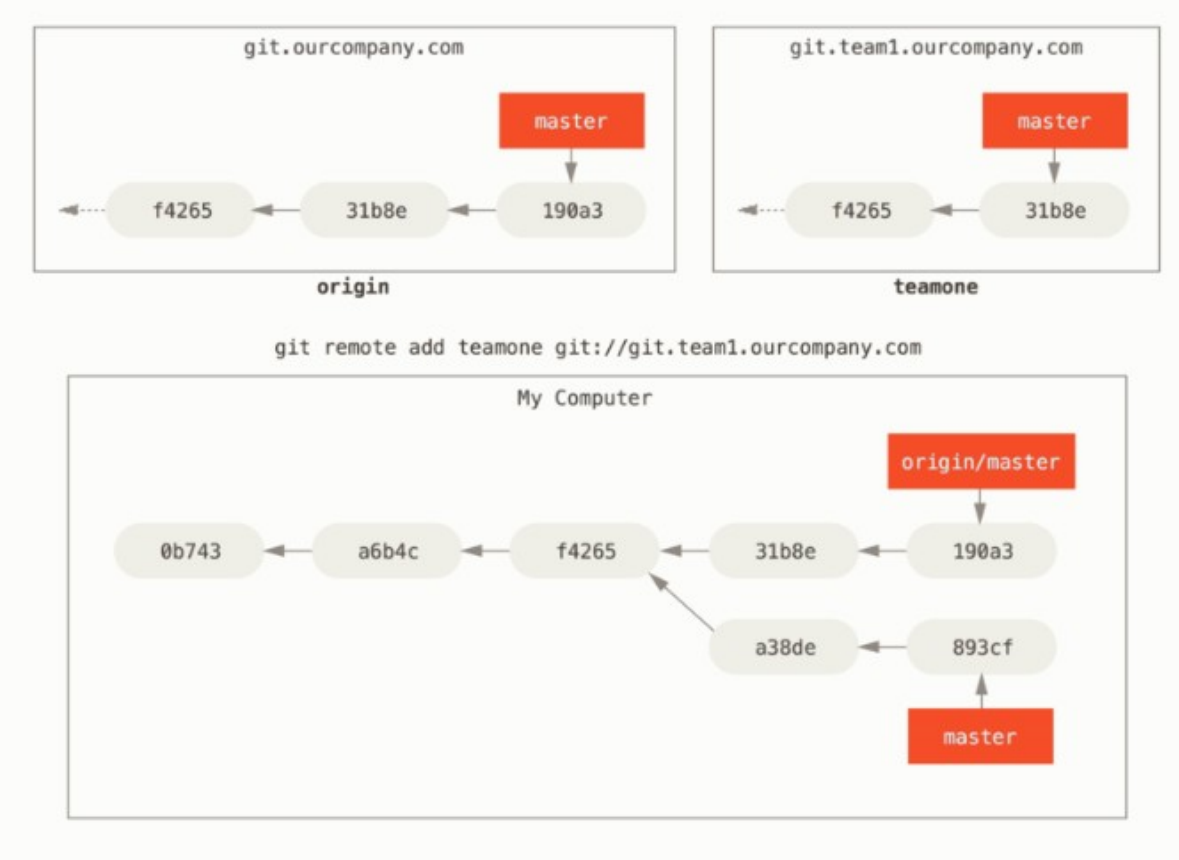


Figure 33. Añadiendo otro servidor como remoto

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.

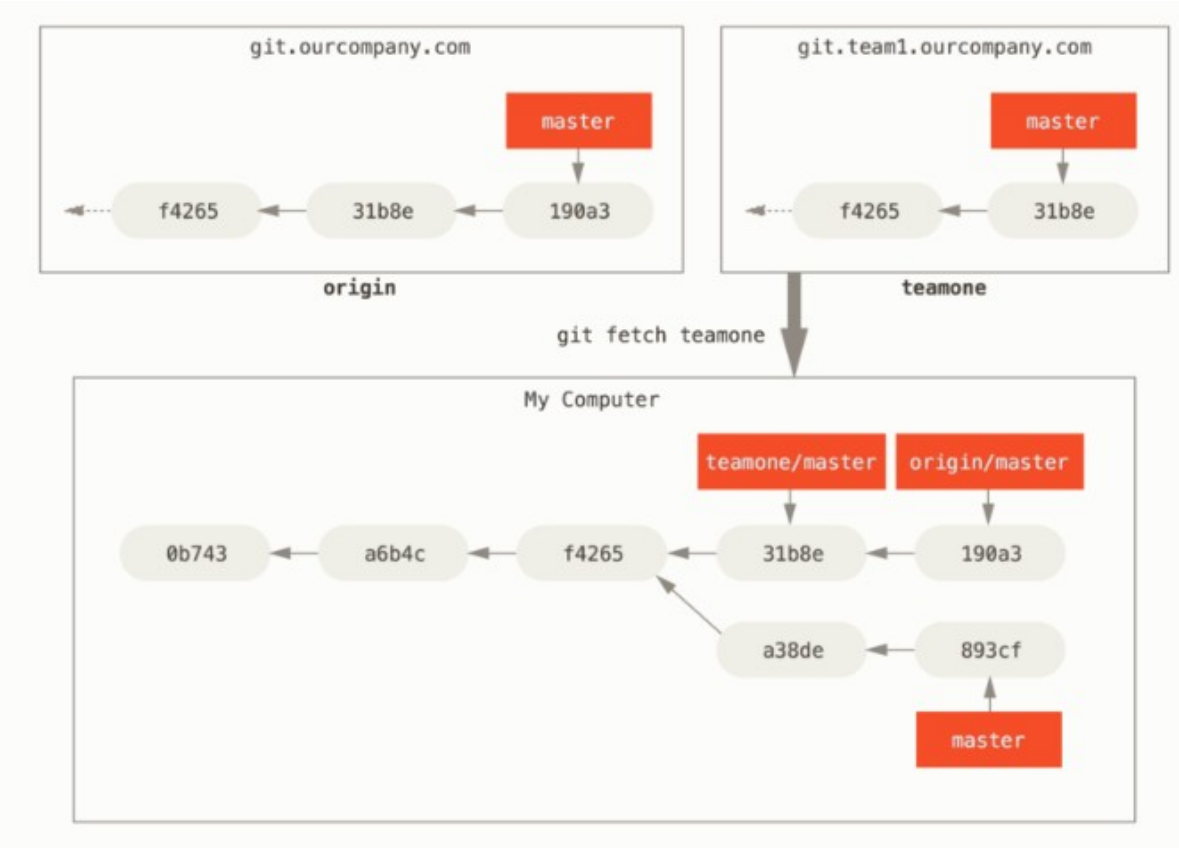


Figure 34. Seguimiento de la rama remota a través de teamone/master

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 desees 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.

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.
```

Si quieres ver las ramas de seguimiento que tienes asignadas, 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 que no tienes del servidor, 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,

```
$ 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.

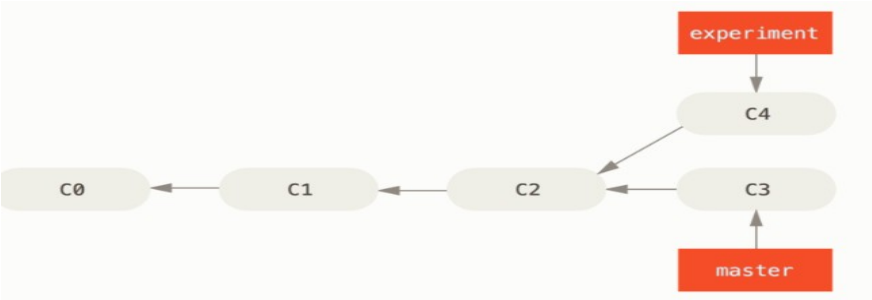


Figure 35. El registro de confirmaciones inicial

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).

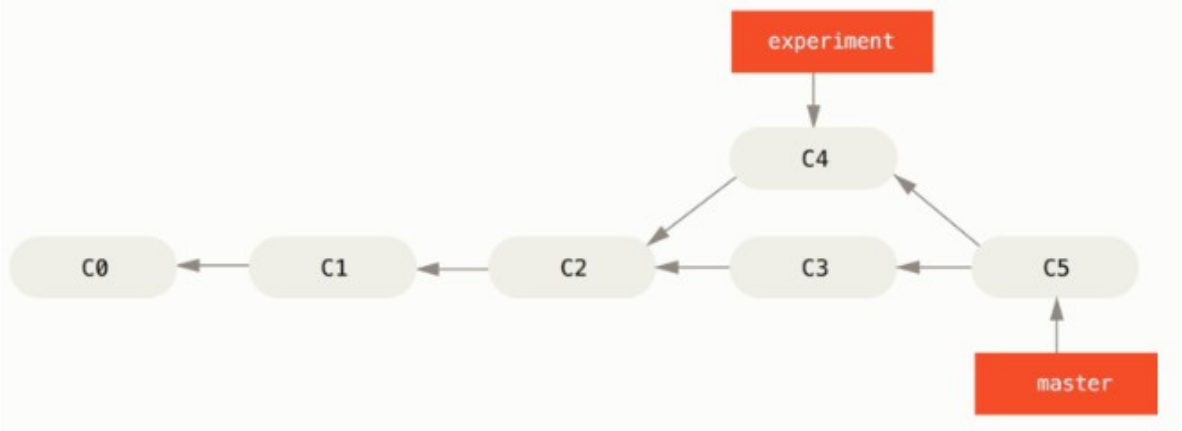


Figure 36. Fusionar una rama para integrar el registro de trabajos divergentes

Sin embargo, también hay otra forma de hacerlo: puedes capturar 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 capturar 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 quieres 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 que la rama de donde quieres reorganizar, y finalmente, vuelva a aplicar ordenadamente los cambios.

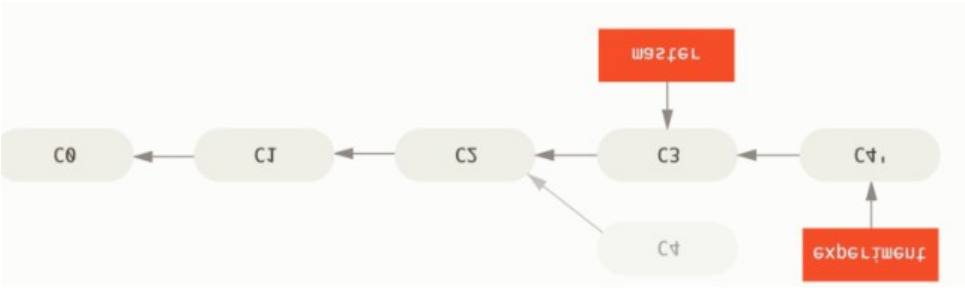


Figure 37. Reorganizando sobre C3 los cambios introducidos en C4

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
```

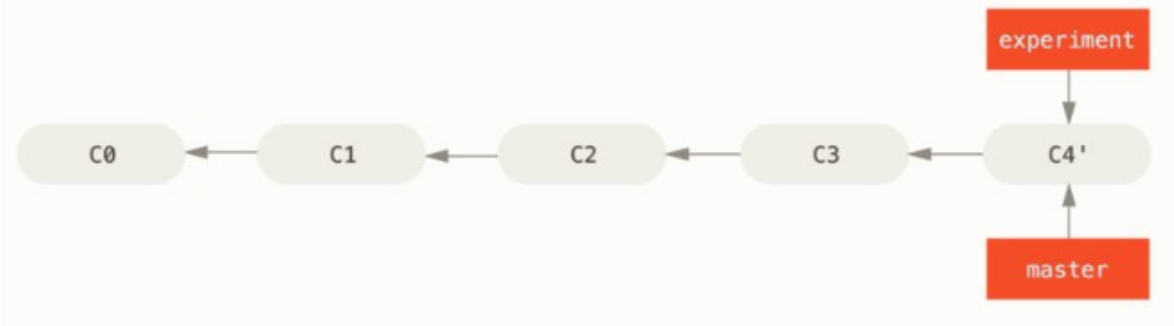


Figure 38. Avance rápido de la rama master

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 no 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. Después, 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.

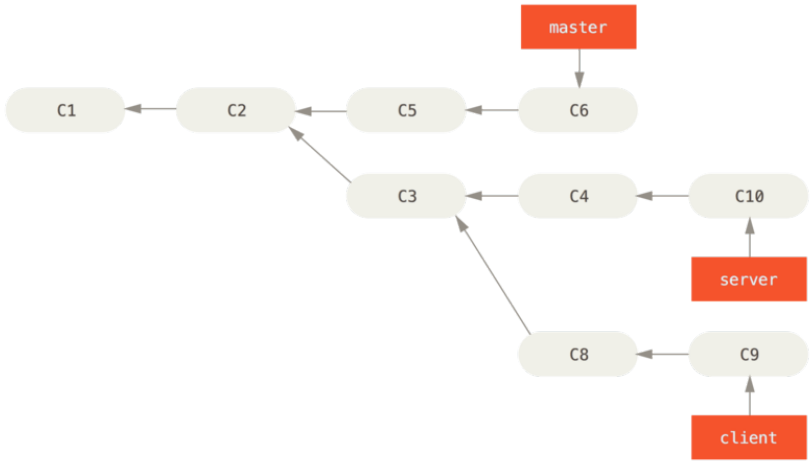


Figure 39. Un historial con una rama puntual sobre otra rama puntual

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 aun 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 aplicalos en la rama master”. Puede parecer un poco complicado, pero los resultados son realmente interesantes.

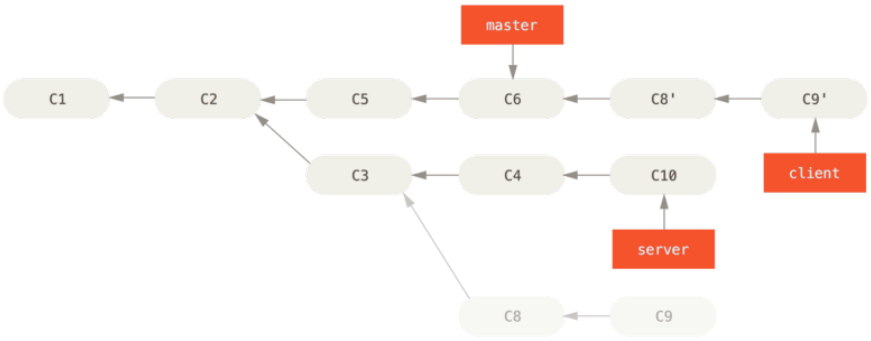
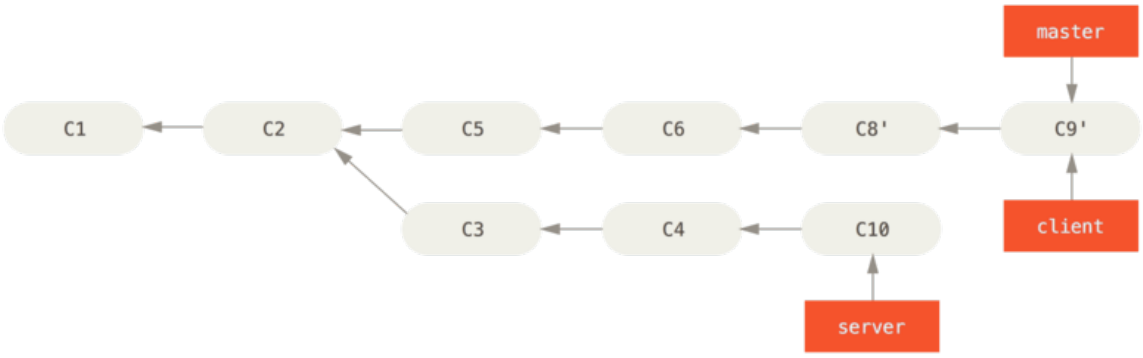


Figure 40. Reorganizando una rama puntual fuera de otra rama puntual

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
```



igure 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.

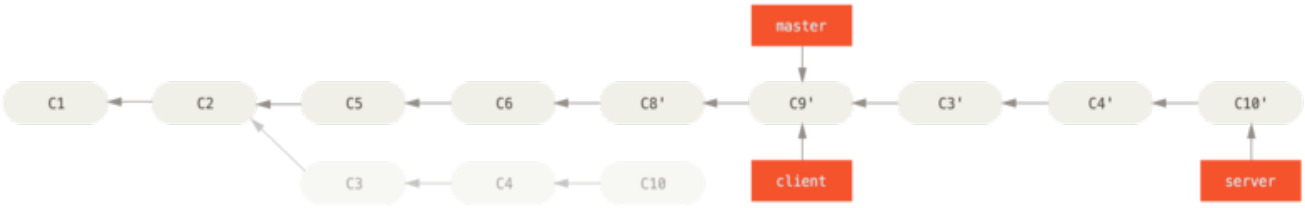


Figure 42. Reorganizando la rama server sobre la rama master

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
```

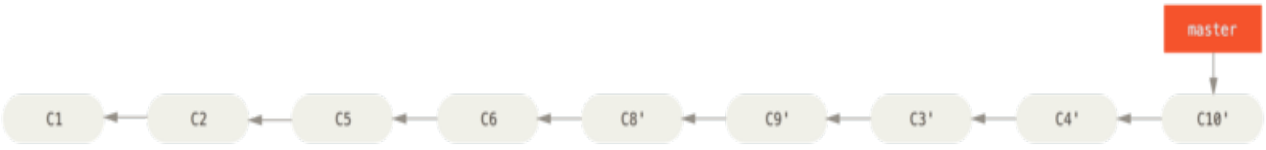


Figure 43. Historial final de confirmaciones de cambio

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 tremendamente 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:

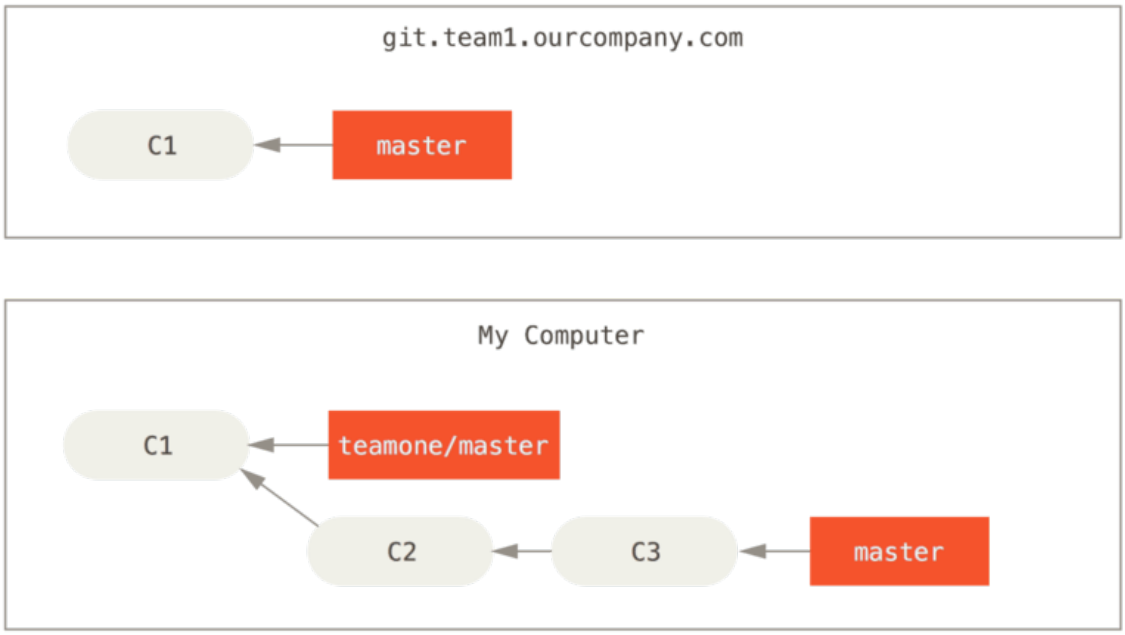


Figure 44. Clonar un repositorio y trabajar sobre él

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:

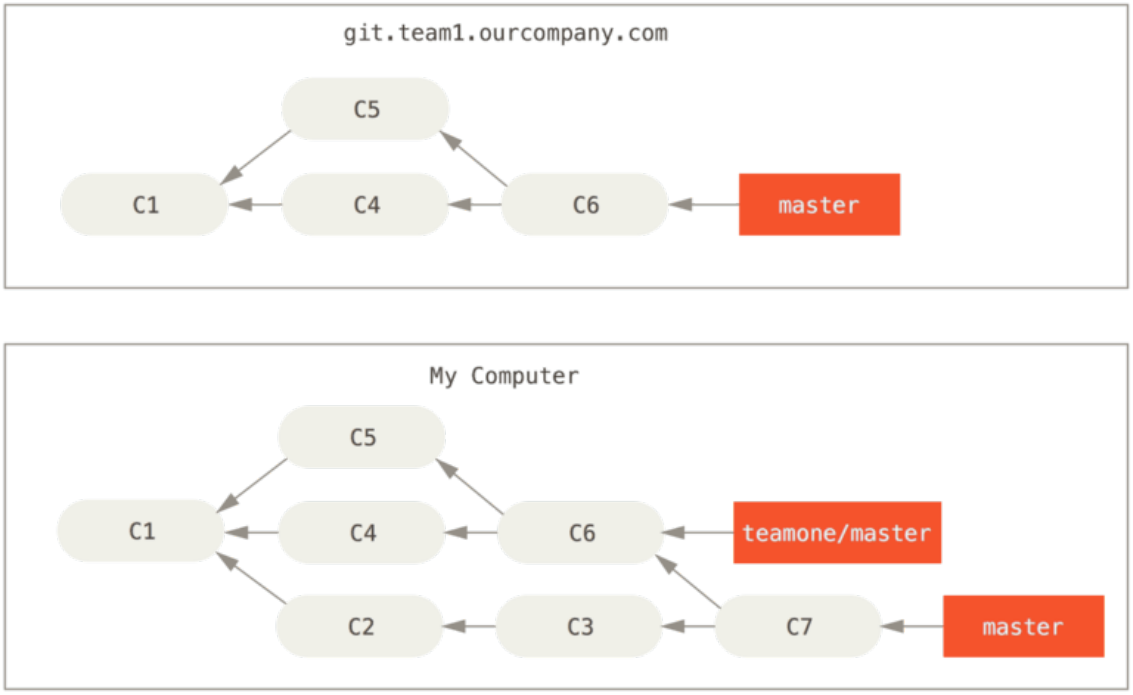


Figure 45. Traer (fetch) algunas confirmaciones de cambio (commits) y fusionarlas (merge) sobre tu trabajo

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.

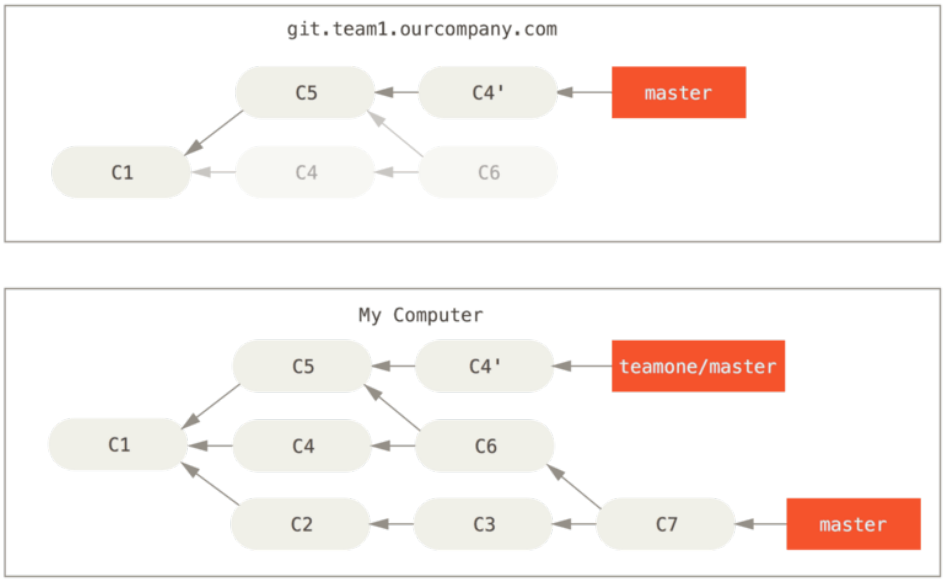


Figure 46. Alguien envió (push) confirmaciones (commits) reorganizadas, abandonando las confirmaciones en las que tu habías basado tu trabajo

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:

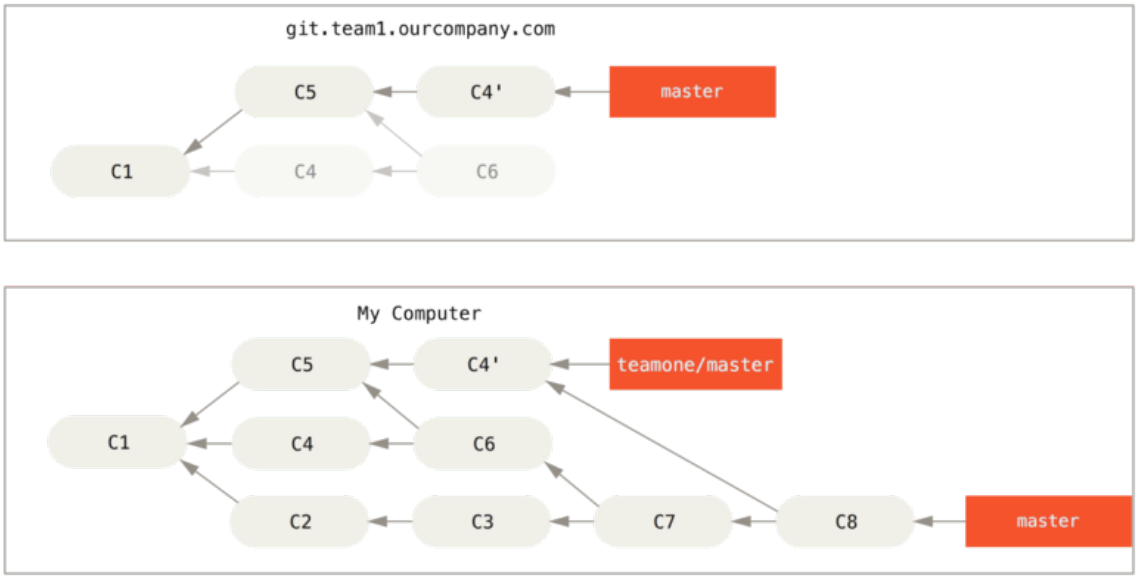


Figure 47. Vuelves a fusionar el mismo trabajo en una nueva fusión confirmada

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 envió (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..

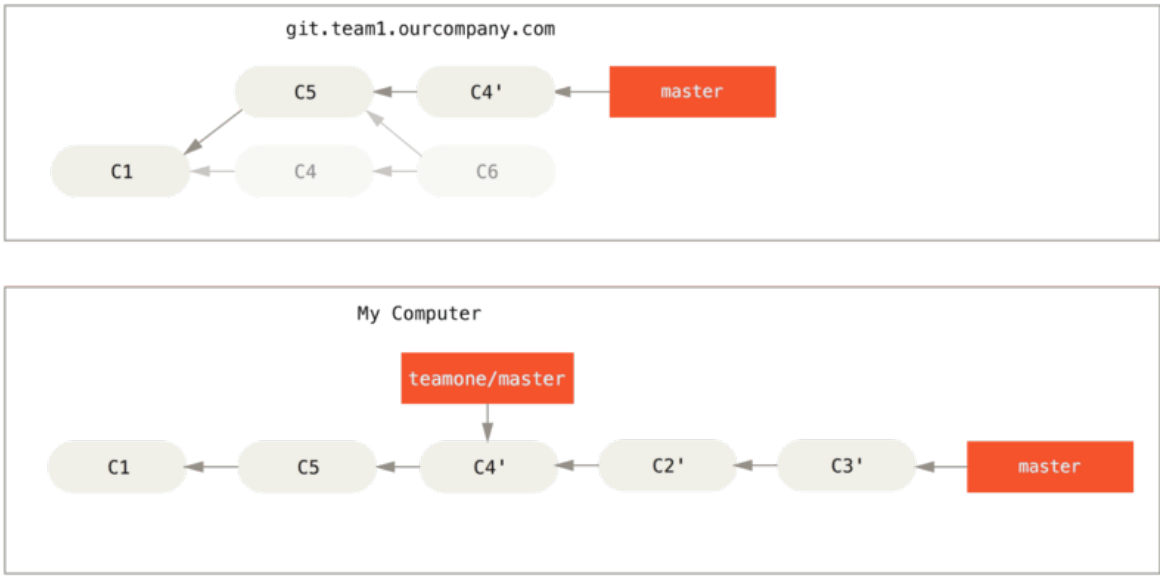


Figure 48. Reorganizar encima de un trabajo sobreescrito reorganizado.

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 estarían 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 sólo 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 fué como ocurrió y el repositorio debería tener un registro de esto para la posteridad.

La otra forma de verlo puede ser 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 si es mejor 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 aún no has compartido, antes de enviarlo a algún lugar; pero nunca reorganizar nada que ya haya sido compartido.

GITHUB

Si es nuestra primera vez utilizando esta plataforma y queremos desde nuestro **git** subir un repositorio a **github** debemos de clickear en **New**, una vez allí colocar el nombre, alguna que otra descripción de lo que vamos a subir, si queremos que sea público o privado,etc. Una vez que esta todo como nos parezca, damos click en **Create Repository.**

Nos saldrán varias líneas de comandos como:

...or create a new repository on the comand line

Nos indica cada uno de los pasos que debemos de hacer para poder mandar desde nuestro repositorio local a el repositorio que hemos creado en **github** y enlazarlos (recomendado cuando empiezas desde 0).

...or push an existing repository from the command line

Este caso hace referencia a que ya tienes un repositorio local y solamente deseas mandarlo a la plataforma.

Para ello deberás de tipear los siguientes comandos en el bash:

git remote add origin https://github.com/tucanal/nombre-repo.git

git push -u origin master

Es cuestión de refrescar la página de github y podrás ver como se ha subido nuestro repositorio.

CLONE

¿Qué pasaría si por alguna razón nos vamos a otra computadora y en esta creamos alguna carpeta y necesitamos colocar en ella nuestro trabajo que hemos estado realizando en el repositorio?

Simplemente yo puedo clonar ese proyecto, seleccionamos simplemente con un click en **"Clone or download"** y una vez allí copiamos la dirección. Nos colocamos en la carpeta deseada para hacer la clonación y tipeamos el siguiente comando:

git clone https://github.com/tucanal/repositorio.git

Una vez que se ha clonado correctamente todo, podemos realizar algún cambio, es decir, modificamos el contenido de algún archivo, lo agregamos al stage y lo commiteamos.

Logrado esto, deberemos colocar en el bash el siguiente comando para que los cambios que hemos realizados desde este nuevo repositorio local al de github:

git push origin master

Si refrescamos nuestro repositorio, podremos observar los cambios que hemos realizado (recordar que hay que ingresar contraseña y usuario para poder realizar estos cambios desde otro repositorio local).

PULL

En la plataforma de **github** podemos agregar, modificar o eliminar archivos del repositorio situado allí, aunque no es recomendado o no muy utilizado, dado que es recomendable que desde nuestro repositorio local remoto debemos de hacer todas estas modificaciones para luego subirlo.

Pero puede surgir algún improvisto, por lo tanto, para poder hacer un **pull** correcto esto es lo que hay que hacer:

Nos situamos en la plataforma de github (en nuestro repositorio) creamos un nuevo archivo, lo commiteamos Antes de confirmar, veremos que hay 2 opciones para dar click el de

- **Commit directly to the master branch:** Es decir que el commit que estamos a punto de confirmar vaya directamente a nuestra rama principal.
- **Create a new Branch for this commit and start a pull request** : Crear una rama distinta y luego utilizar el comando **pull request** para traer los cambios al repositorio local remoto.

Ahora es cuando hay que traer los cambios, para ello **git** nos ofrece 2 comandos:

Git fetch : donde nosotros traeremos los cambios que hemos realizado en la plataforma a nuestro repositorio local y luego deberemos de hacer un **git merge.**

Git pull : directamente traerá los cambios realizados en la plataforma a nuestro repositorio local remoto (a la rama master) y los unirá.

Hacemos un **git pull** y veremos las alteraciones en nuestro repositorio local.

+ Aportes -> Investigar un poco con respecto a las secciones **raw, blame y history** en github.

FORK

En muchas ocasiones podemos ver en github repositorios que son de nuestro agrado y donde nosotros queremos tener una copia y hacerle algunas modificaciones, para ello github nos ofrece la implementación de **fork.**

Buscamos algún repositorio que nos parezca interesante y yo quisiera tener ese proyecto en mi repositorio debería de hacer un **fork** dando click en **Fork** situado en la parte superior derecha de github. Una vez hecho comenzará el **forking** y esto sirve porque podremos trabajar con los archivos de ese proyecto y los cambios que hagamos no irán al repositorio original del autor de ese proyecto, si no a nuestra copia local.

Entonces solo resta clonar ese repositorio que esta en nuestro canal para poder trasladarlo a alguna carpeta deseada y seguir trabajando desde nuestro repositorio local.

PULL REQUEST

Para entender esto podes por ejemplo crear una cuenta adicional en **github** para que puedas desde allí hacer un **Fork** de alguno de los repositorios de tu cuenta original, clonarlo en alguna carpeta y en algún archivo que nosotros notamos que podría mejorarse algo, procedemos a realizar algún cambio, lo agregamos al stage y lo commiteamos (todo esto desde nuestro repositorio local remoto) y luego mandamos los cambios a el forking que hicimos con un **git push origin master** .

Ese cambio que hemos hecho queremos que sea visto por el creador del repositorio original (en este caso nosotros, desde nuestra cuenta original), debemos hacérselo notar puesto que en el repositorio original los cambios que hicimos previamente, no se verán reflejados.

Para hacer el pull request nos dirigiremos a la solapa de **Pull requests** allí daremos click en **new pull request**, veremos una ventana a modo de resumen en donde se reflejarán los cambios que hemos hecho nosotros en comparación al repositorio original (el código original, mejor dicho). Daremos click en **Create pull request** donde veremos el asunto (colocamos algún mensaje global) y más abajo tenemos suficiente lugar para poder explayarnos en mencionar el porque ese cambio que hemos realizado nosotros, sería considerado como algo que a el repositorio original le vendrían bien agregarlo.

El autor del repositorio verá en sus pull requests el mensaje que le hemos enviado, para que lo pueda observar y si lo considera realizar el cambio pertinente (además de poder responderle al usuario que le ha propuesto ese cambio).

Lo bueno de todo esto es que si el usuario original considera que esta modificación es buena y no genera conflictos con la rama maestra de su repositorio local remoto, puede clickear en **Merge pull request** y de esta manera sumará a su repositorio los cambios que hizo un usuario (en modo de ayuda).