

Fork

Sirve para crear una copia de un repositorio en nuestra cuenta de usuario. Ese repositorio copiado será un clon del repositorio desde el que se hace el fork, pero a partir de entonces el fork vivirá en un espacio diferente y podrá evolucionar de manera distinta, a tu propio cargo.

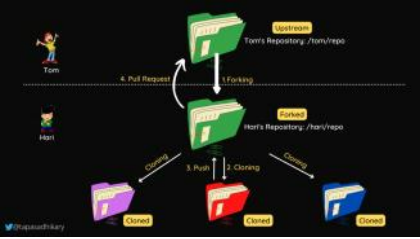
El fork se entiende como una rama externa de un repositorio, colocando esa rama en un nuevo repositorio controlado por otros usuarios. Una vez hecho el fork existirán dos repositorios distintos. Inicialmente uno era copia exacta del otro, pero a medida que se va va desarrollando y publicando cambios en uno u otro repo, ambos repositorios podrán tender a ser tan distintos como queramos cada uno de los equipos de desarrollo que los mantengan.

¿Para qué necesito un Fork?


Un fork es una copia de un repositorio, pero ¿por qué no lo llamamos el repositorio que queremos copiar y listo?

Si hacemos un clon normal de un repositorio, el espacio en GitHub de ese clon seguirá asociado al repositorio que has clonado. De este modo, si realizamos cambios sobre el clon y los quieres publicar en GitHub, probablemente no los podrás subir.

Obviamente, si donamos un repositorio que era de nosotros, podrás realizar cambios en local y subirlos a GitHub siempre. Pero si el repositorio era de otro desarrollador y no tenemos permisos de escritura sobre él, entonces no es posible subir cambios, porque GitHub no lo permitirá. Para estos casos después de hacer un fork sigue realizar un pull request.



Stach



El comando `git stash` guarda el trabajo actual del Staging en una lista diseñada para ser temporal llamada Stash, para que pueda ser recuperado en el futuro.

Para agregar los cambios al stash se utiliza el comando

git stash

Podemos poner un mensaje en el stash, para así diferenciarlos en `git stash list` por si tenemos varios elementos en el stash. Esto con:

```
git stash save "mensaje identificador del elemento del stashed"
```

a) Obtener elementos del stash

El stashed se comporta como una Stach de datos comportándose de manera tipo LIFO (del inglés Last In, First Out, «último en entrar, primero en salir»), así podemos acceder al método pop.

El método pop recuperará y sacará de la lista el último estado del stashed y lo insertará en el staging area, por lo que es importante saber en qué branch te encuentras para poder recuperarlo, ya que el stash será agnóstico a la rama o estado en el que te encuentres. Siempre recuperará los cambios que hiciste en el lugar que lo llamas.

b) Para recuperar los últimos cambios desde el stash a tu staging área utiliza el comando:

```
git stash pop
```

c) Para aplicar los cambios de un stash específico y eliminarlo del stash:

```
git stash pop stash@{<num_stash>}
```

Para retomar los cambios de una posición específica del Stash puedes utilizar el comando

```
git stash apply stash@{<num_stash>}
```

Donde el `<num_stash>` lo obtienes desde el `git stash list`

d) Listado de elementos en el stash

Para ver la lista de cambios guardados en Stash y así poder recuperarlos o hacer algo con ellos podemos utilizar el comando

```
git stash list
```

Consideraciones

El cambio más reciente (al crear un stash) SIEMPRE recibe el valor 0 y los que estaban antes aumentan su valor.

Al crear un stash tomará los archivos que han sido modificados y eliminados. Para que tome un archivo creado es necesario agregarlo al Staging Area con `git add nombre_archivo` con la intención de que git tenga un seguimiento de ese archivo, o también utilizando el comando `git stash -u` (que guardará en el stash los archivos que no estén en el staging).

Al aplicar un stash este no se elimina, es buena práctica eliminarlo

Clean

Limpia el árbol de trabajo eliminando recursivamente los archivos que no están bajo control de versiones, empezando por el directorio actual.

Normalmente, sólo se eliminan los archivos desconocidos para Git, pero si se especifica la opción `-x`, también se eliminan los archivos ignorados. Esto puede ser útil, por ejemplo, para eliminar todos los productos de compilación.

Si se especifica cualquier argumento opcional «especificación de ruta», sólo se verán afectadas las rutas que concidan con la especificación de ruta.

```
git clean -x -f -- --exclude=/git-demos/ --exclude=docs/
```

OPCIONES PARA AGREGAR AL COMANDO

-d

Normalmente, cuando no se especifica, `git clean` no recurrirá a directorios sin seguimiento para evitar eliminar demasiado. Especifica `-d` para que también recurra a esos directorios. Si se especifica «especificación de ruta», `-d` es irrelevante, todos los archivos no rastreados que concidan con las rutas especificadas (con excepciones para los directorios git anidados mencionados en `--force`) serán eliminados.

-f

--force

Si la variable de configuración de `git clean`, `Require Force` no está establecida a `false`, `git clean` se negará a borrar archivos o directorios a menos que se le dé `-f` o `-F`. Git se negará a modificar repositorios git anidados no seguidos (directorios con un subdirectorio git) a menos que se indique un segundo `-f`.

-i

--interactive

Muestra lo que se haría y limpia los archivos de forma interactiva. Ver "Modo interactivo" para más detalles.

-n

--dry-run

No elimina nada, sólo muestra lo que se haría.

-q

--quiet

Ser silencioso, sólo informar de los errores, pero no los archivos que se eliminan con éxito.

-X

Elimina sólo los archivos ignorados por Git. Esto puede ser útil para reconstruir todo desde cero, pero manteniendo los archivos creados manualmente.

Cherry-pick

git cherry-pick es un comando poderoso que permite seleccionar confirmaciones de Git arbitrarias por referencia y agregarlas al HEAD de trabajo actual.

La selección de cherry es el acto de seleccionar una confirmación de una rama y aplicarla a otra. **git cherry-pick** puede ser útil para deshacer cambios. Por ejemplo, supongamos que una confirmación se realiza accidentalmente en la rama incorrecta. Puede cambiar a la rama correcta y seleccionar la confirmación donde debería pertenecer.

¿Cuándo usar git cherry pick?

Usando `git cherry-pick` es un herramienta útil pero no siempre una buena práctica. La selección selectiva puede causar compromisos duplicados y muchos escenarios en los que la selección selectiva funciona, en su lugar se prefieren las fusiones tradicionales. Dicho esto, `git cherry-pick` es una herramienta útil para algunos escenarios.

Colaboración en equipo.

A menudo, un equipo encontrará miembros individuales trabajando en o alrededor del mismo código. Tal vez una nueva característica del producto tenga un componente de backend y frontend.

Puede haber algún código compartido entre dos sectores de productos. Tal vez el desarrollador del backend crea una estructura de datos que el frontend también necesitará utilizar.

El desarrollador frontend podría usar **git cherry-pick** para elegir la confirmación en la que se creó esta estructura de datos hipotética. Esta elección permitiría al desarrollador frontend continuar el progreso en su lado del proyecto.

Correcciones de errores

Cuando se descubre un error, es importante entregar una solución a los usuarios finales lo más rápido posible. Para un escenario de ejemplo, digamos que un desarrollador ha comenzado a trabajar en una nueva característica.

Durante el desarrollo de esa nueva función, identifican un error preexistente. El desarrollador crea una confirmación explícita para corregir este error. Esta nueva confirmación de parche se puede seleccionar directamente en la rama main para corregir el error antes de que afecte a más usuarios.

Deshacer cambios y restaurar confirmaciones perdidas

A veces, una `feature-rama` puede volverse obsoleta y no fusionarse con `main`. A veces, una solicitud de extracción puede cerrarse sin fusionarse. Git nunca pierde esos compromisos y, a través de comandos como `git revert`, se pueden encontrar y volver a la vida `git loggit revert`.

Rebase

¿Qué es git rebase?

Es un comando utilizado para el proceso de mover o combinar una secuencia de confirmaciones a una nueva confirmación base. El rebase es más útil y fácil de visualizar en el contexto de un flujo de trabajo de forks de características. El proceso general puede visualizarse como sigue:

Uso

La razón principal para usar rebase es mantener una historia lineal del proyecto. Por ejemplo, considera una situación en la que la rama principal ha progresado desde que empezaste a trabajar en una rama de características. Quieres obtener las últimas actualizaciones de la rama principal en tu rama de características, pero quieres mantener limpio el historial de tu rama para que parezca que has estado trabajando en la rama principal más reciente. Esto da el beneficio posterior de una fusión limpia de su rama de características de nuevo en la rama principal. ¿Por qué queremos mantener un "historial limpio"? Los beneficios de tener un historial limpio se hacen tangibles al realizar operaciones Git para investigar la introducción de una regresión. Un escenario más real sería:

Se identifica un bug en la rama principal. Una característica que funcionaba correctamente ahora está rota.

Un desarrollador examina la historia de la rama principal usando `git log`, porque gracias a la "historia limpia" el desarrollador es rápidamente capaz de razonar sobre la historia del proyecto.

El desarrollador no puede identificar cuándo se introdujo el fallo usando `git log`, así que ejecuta un `git bisect`.


Como el historial de git está limpio, `git bisect` tiene un conjunto refinado de confirmaciones para comparar cuando busca la regresión. El desarrollador encuentra rápidamente la confirmación que introdujo el error y puede actuar en consecuencia.

Aprende más sobre `git log` y `git bisect` en sus páginas de uso individuales.

Tienes dos opciones para integrar tu característica en la rama principal: fusionar directamente o hacer un rebase y luego fusionar. La primera opción da como resultado una fusión a tres bandas y una confirmación de fusión, mientras que la segunda da como resultado una fusión rápida y un historial perfectamente lineal. El siguiente diagrama muestra cómo hacer un rebase a la rama principal facilita una fusión rápida.

Git rebase: Fork en la rama principal

Rebasarse es una forma común de integrar los cambios de la rama principal en tu repositorio local. Incorporar los cambios de la rama principal con `Git merge` resulta en una confirmación superflua cada vez que quieres ver cómo ha progresado el proyecto. Por otro lado, hacer rebase es como decir: "Quiero basar mis cambios en lo que ya ha hecho todo el mundo".



Pull Request

Los pull request se pueden usar junto con el flujo de trabajo de rama de función, el flujo de trabajo de GitFlow o el flujo de trabajo de un fork. Pero un pull request requiere dos ramas distintas o dos repositorios distintos, por lo que no funcionarán con el flujo de trabajo centralizado. El proceso se resume en lo siguiente:

- Un desarrollador crea la característica en una rama dedicada en su repositorio local.
- El desarrollador envía la rama a un repositorio público.
- El desarrollador presenta una solicitud de extracción a través de la sección de pull request.
- El resto del equipo revisa el código, lo discute y lo modifica.
- El mantenedor del proyecto fusiona la función en el repositorio oficial y cierra la solicitud de extracción.

El resto de esta sección describe cómo se pueden aprovechar las solicitudes de incorporación de cambios en diferentes flujos de trabajo de colaboración.

