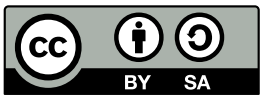
Gestión de ramas con Git

Antonio García Domínguez
antonio.garciadominguez@uca.es

24 de septiembre de 2014



Índice

1. Gestión básica de ramas

Las revisiones enviadas a un repositorio Git forman un grafo acíclico dirigido, en el que podemos tener líneas de revisiones que se dividen a partir de un cierto punto en múltiples ramas. Estas ramas posteriormente pueden reunirse opcionalmente, aunque no es estrictamente necesario.

Siempre estamos trabajando con una rama: por defecto, Git crea siempre la rama `master`, considerada normalmente como la rama principal de desarrollo (`trunk` para aquellos que conozcan Subversion). Podemos ver en qué rama estamos con:

```
git branch
```

Obtendremos una salida como:

```
* master
```

Esta salida indica que nos hallamos actualmente trabajando sobre la punta de la rama `master`, con lo que cualquier revisión que vayamos enviando no sólo creará el objeto correspondiente, sino que además hará avanzar el puntero `master` además del `HEAD`. Puede que no estemos sobre la punta de ninguna rama si hemos movido el `HEAD` manualmente (después veremos cómo). En dicho caso veríamos algo así:

```
* (no branch)
  master
```

Hay que tener cuidado: si creamos nuevas revisiones sin que sean alcanzables por una rama, estas revisiones no son alcanzables de forma normal y serán recolectadas como basura tras un período de gracia de 30 días por defecto. Podemos marcar la revisión actual como la punta de una rama (sin llegar a cambiarnos a ella) con:

```
git branch -a nombrerama
```

Por otro lado, podemos eliminar una rama (es decir, la referencia a su punta) con:

```
git branch -d nombrerama
```

Sin embargo, hay que tener cuidado en ciertos casos. Borrar la referencia a una rama que ya ha sido reunida con alguna otra no tiene problema, ya que sus revisiones son alcanzables desde la otra rama, como se ve en el caso de `develop` en la figura 1 en la página siguiente. Sin embargo, si aún no se ha hecho esto, como en 2 en la página siguiente, podríamos acabar perdiendo las revisiones de dicha rama, al quedar inalcanzables por toda referencia.

Por ello, `git checkout -d develop` fallaría en el segundo caso, y si de verdad quisiéramos eliminar esa rama y descartar todas sus revisiones, sustituiríamos la opción `-d` por `-D`. Esto es útil, por ejemplo, para descartar una rama que hayamos visto improductiva por completo sin tener que reunirla.

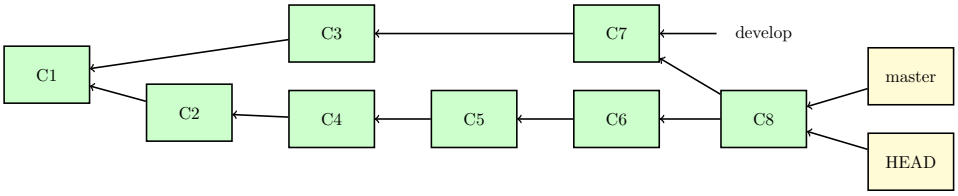


Figura 1: Situación no problemática al borrar la rama `develop`

Figura 2: Situación problemática al borrar la rama `develop`

Se recuerda que para ver de forma cómoda el grafo de revisiones desde la revisión actual, podemos usar `gitk`, y para ver el grafo completo, se puede utilizar `gitk --all`.

2. Cambio entre ramas

Para cambiar entre ramas, o en general mover el `HEAD` de forma no destructiva a cualquier revisión, usaremos la orden `git checkout <treeish>`. Así, cambiar a la rama `develop` es tan simple como ejecutar `git checkout develop`. Esto actualizará tanto el índice como el directorio del trabajo al aspecto que tenían en dicha revisión. Si tenemos cambios locales sobre ficheros cuyo contenido es distinto en la revisión en la que estamos trabajando, `git checkout` informará del error y abortará su ejecución.

Para solventar este problema, tenemos dos opciones:

1. Guardar los cambios aparte mediante `git stash`, una nueva orden introducida en Git v1.5.3, antes de cambiar de revisión. Esta orden es puramente de conveniencia: internamente usa operaciones normales y corrientes de Git.

   Las entradas del *stash* forman una pila, y se pueden gestionar de forma muy flexible. Las órdenes disponibles son:

   - `git stash`: guarda los cambios sobre el índice y el directorio de trabajo aparte en una nueva entrada, y deja el índice y el directorio de trabajo con los contenidos de `HEAD`.
   - `git stash apply [stash@{N}]`: toma la entrada N y aplica sus cambios. Por defecto, utiliza $N = 0$, es decir, el tope de la pila.
   - `git stash drop stash@{N}`: elimina la entrada N.
   - `git stash pop`: aplica el tope de la pila y lo retira.
   - `git stash clear`: elimina todas las entradas.
   - `git stash list`: lista todas las entradas.

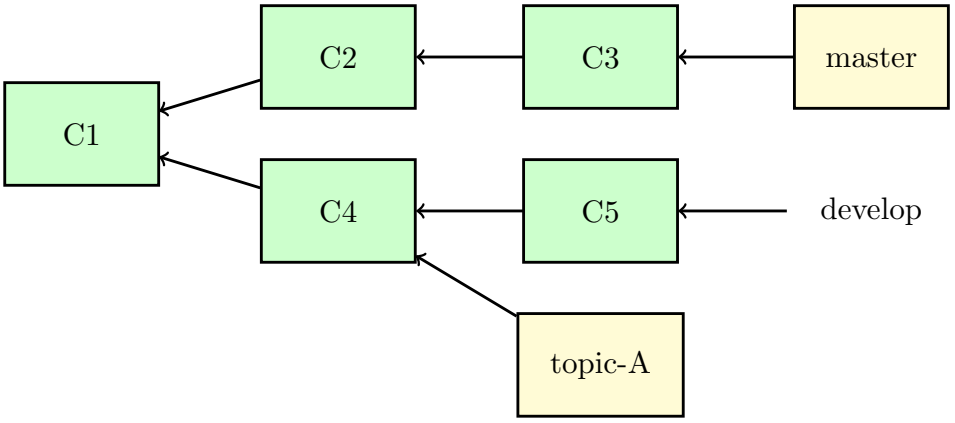


Figura 3: Estado original de las ramas a reunir

2. Obligar con la opción `-m` de `git checkout` a hacer una reunión (veremos posteriormente en qué consisten) entre el estado del índice, la revisión en que se estaba y la revisión destino. Esto puede generar conflictos: los veremos posteriormente.

Se recuerda que al cambiar a una revisión que no esté en la punta de una rama (Git nos avisará con algo del estilo de «moving to X which isn't a local branch»), las posteriores revisiones que enviemos no serán alcanzables desde ninguna de las ramas. Deberíamos de añadir una rama en donde estemos para evitar esto, y cambiar a dicha rama. Podríamos hacerlo así para la revisión con *commitish* `HEAD 2`

```
git checkout HEAD~2
git branch -a otrarama
git checkout otrarama
```

Existe un atajo:

```
git checkout -b otrarama HEAD~2
```

3. Reunión de ramas

Ya sabemos ramificar nuestro árbol de revisiones. Ahora hemos de averiguar cómo reunir esas ramas de nuevo en una sola. Para ello, Git nos ofrece dos posibilidades. Si nadie tiene aún nuestras revisiones, no hay problema en usar cualquiera de las dos, pero de lo contrario, lo mejor es no utilizar la segunda solución aquí ofrecida, o nos buscamos un buen disgusto. Usaremos como estado inicial del repositorio el de la figura 3.

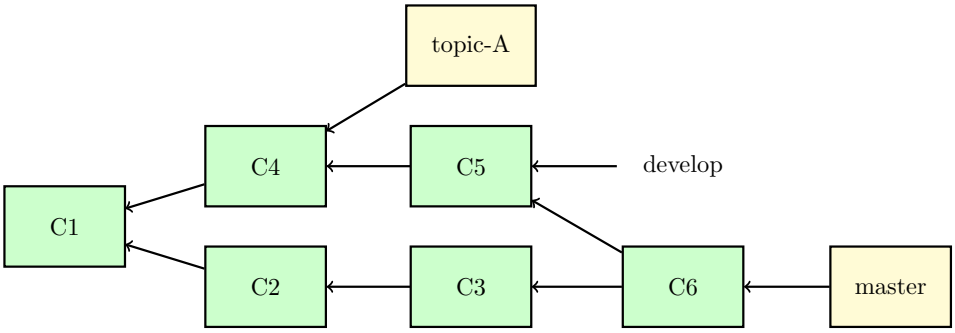


Figura 4: Estado tras `git merge develop` con `HEAD` en `master`

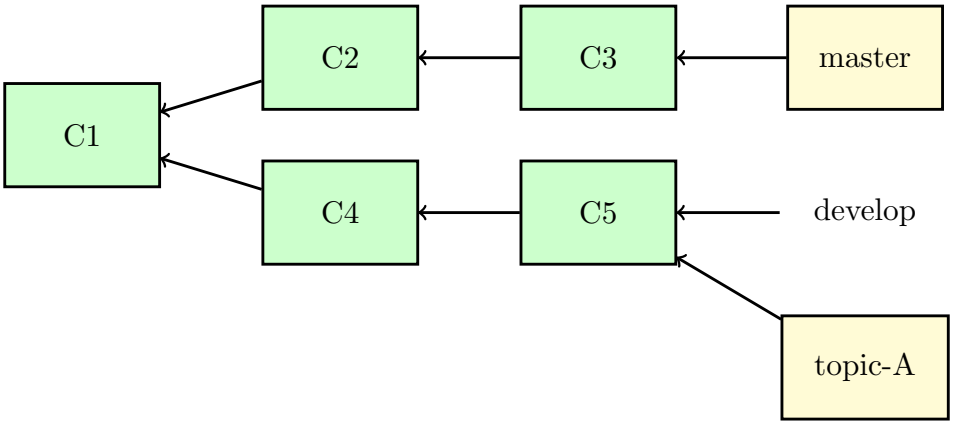


Figura 5: Estado tras `git merge develop` con `HEAD` en `topic-A`

3.1. Añadir una revisión que reúne a las dos

Esta primera solución, la más segura con diferencia, consiste en crear una nueva revisión que reunirá los cambios de una rama con los de la otra, tal y como se ve en la figura 4 en la página anterior. Es posible que aparezca un conflicto entre dos cambios que se hayan hecho. Supongamos que estamos en la rama `master` y que estamos reuniéndola con la rama `develop`. Si en nuestra rama hemos añadido al final la línea «D» y en la otra la línea «1», se producirá un conflicto, que nos dejará el fichero `f` con este aspecto:

```
A
C
B
<<<<<<< HEAD:f
D
=======
1
>>>>>>> develop:f
```

La primera parte antes de «=======» indica el contenido en conflicto de la versión de la rama `master` del fichero `f`. La otra mitad se ocupa de la parte del contenido correspondiente a la rama `develop`.

Otra forma de ver el conflicto es a través de las órdenes:

- `git diff --base`, que muestra las diferencias entre el directorio de trabajo y el ancestro común a las dos ramas que estamos uniendo.
- `git diff --ours` toma como base la copia de la rama en la que estábamos antes de iniciar la reunión.
- `git diff --theirs`, por el contrario, toma como base la copia de la rama con que nos estamos reuniendo.
- `git show :1:f` permite acceder a la copia de `f` del ancestro común a las dos ramas que están siendo reunidas.
- `git show :2:f` es una forma de ver el fichero `f` tal y como estaba en la rama en que estábamos.
- `git show :3:f` muestra cómo estaba `f` en la rama con que nos estamos reuniendo.

Una vez el conflicto quede resuelto, añadiremos los ficheros afectados al índice y crearemos la revisión:

```
git add f
git commit
```

No siempre tendrá por qué crearse una nueva revisión por completo. Si volvemos al ejemplo de la figura 3 en la página 4, veremos que la rama `topic-A` es una antecesora de la rama `develop`. Ejecutando estas órdenes:

```
git checkout topic−A
git merge develop
```

Veremos con toda seguridad el texto «Fast forward» entre la salida de la segunda orden. Esto quiere decir que como `topic-A` era un antecesor de `develop`, sólo ha tenido que adelantarlo hasta donde estaba `develop`, quedando el grafo de revisiones como en la figura 5 en la página 5.

Si posteriormente queremos deshacer el `git merge`, volvemos a tener dos opciones:

1. Si nadie tiene aún la revisión creada, la siguiente orden la elimina por completo. No se debe usar si ya se ha enviado a algún repositorio.

   ```
   git reset −−hard HEAD^
   ```

2. Si ya la tiene alguien, habremos de crear una nueva revisión que deshaga los cambios. `git revert` es capaz de hacerlo si decimos con la opción `-m` cuál de los padres constituye la rama principal, cancelando los cambios de la otra rama.

   Para el caso de la figura 4 en la página 5, esto desharía los cambios introducidos por la rama `develop`:

   ```
   git revert −m 1 HEAD
   ```

   Para revertir los cambios introducidos por la rama `master`, usaríamos:

   ```
   git revert −m 2 HEAD
   ```

3.2. Reescribir varias revisiones en base a otras

Para entender mejor esta opción, comentaremos un caso en el que es particularmente útil. Supongamos que hemos clonado un repositorio de un proyecto de software libre muy activo (como Git) y que hemos desarrollado una nueva funcionalidad. Queremos enviarla a la lista de correo, pero mientras estuvimos haciéndola alguien estuvo tocando esa misma parte. Tenemos que asegurarnos de que nuestras modificaciones se siguen aplicando limpiamente sobre la última versión: es posible que esa persona introdujera cambios que incluyeran a los nuestros o incluso que entraran en conflicto.

Así, lo que queremos es reescribir los cambios introducidos en una rama a partir de otra. Para ello disponemos de la orden `git rebase`. Supongamos que a partir de la situación de la figura 3 en la página 4 ejecutamos o bien:

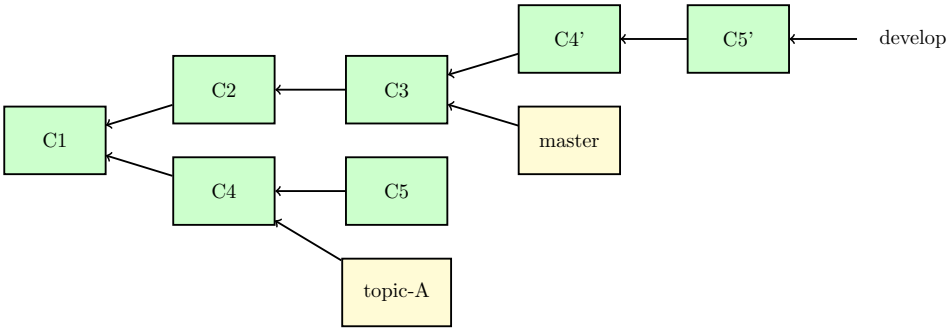


Figura 6: Estado tras `git rebase master` con `HEAD` en `develop`

```
git checkout develop
git rebase master
```

O bien:

```
git rebase master develop
```

En ambos casos obtendremos la situación de la figura 6, en que las antiguas revisiones C4 y C5 se han reescrito en términos de la punta de `master`, C3. Si C2 o C3 incluían parte de los cambios de C4 o C5, puede que C4' o C5' se vean reducidos o eliminados.

También puede que se dé un conflicto, que resolveremos de forma equivalente al caso de `git merge`, salvo por el hecho de que en vez de seguir tras un `git commit`, seguiremos con `git rebase --continue`. También podemos saltarnos la revisión en que nos quedamos parados con `git rebase --skip`, o cancelar por completo el proceso de reescritura y volver al estado original con `git rebase --abort`.

Una opción muy potente de `git rebase` es `--interactive`, que activa un modo interactivo en el que se nos abre un editor y podemos editar una serie de líneas para fusionar, retirar y editar las revisiones a ser reescritas a nuestro antojo. Editaremos un fichero como éste:

```
pick 61fc270 primero
pick e16e7b1 segundo
pick 0749d26 tercero
```

Si cambiamos un «pick» a «squash», reuniremos esa revisión con la anterior. Por el contrario, si usamos «edit» `git rebase` se detendrá en dicha revisión, dejándonos editarla y prepararla como para envío sin hacer `git commit` y continuando el proceso con `git rebase --continue`.

También podemos reordenar las líneas o borrarlas para reordenar o retirar las revisiones correspondientes.

Un uso que tiene esto es para reordenar nuestras revisiones más recientes

que aún no tiene nadie, y enviar un parche lo más limpio posible a una lista de correo. Si quisiéramos reorganizar nuestras últimas 4 revisiones, usaríamos:

```
git rebase --interactive HEAD~4
```