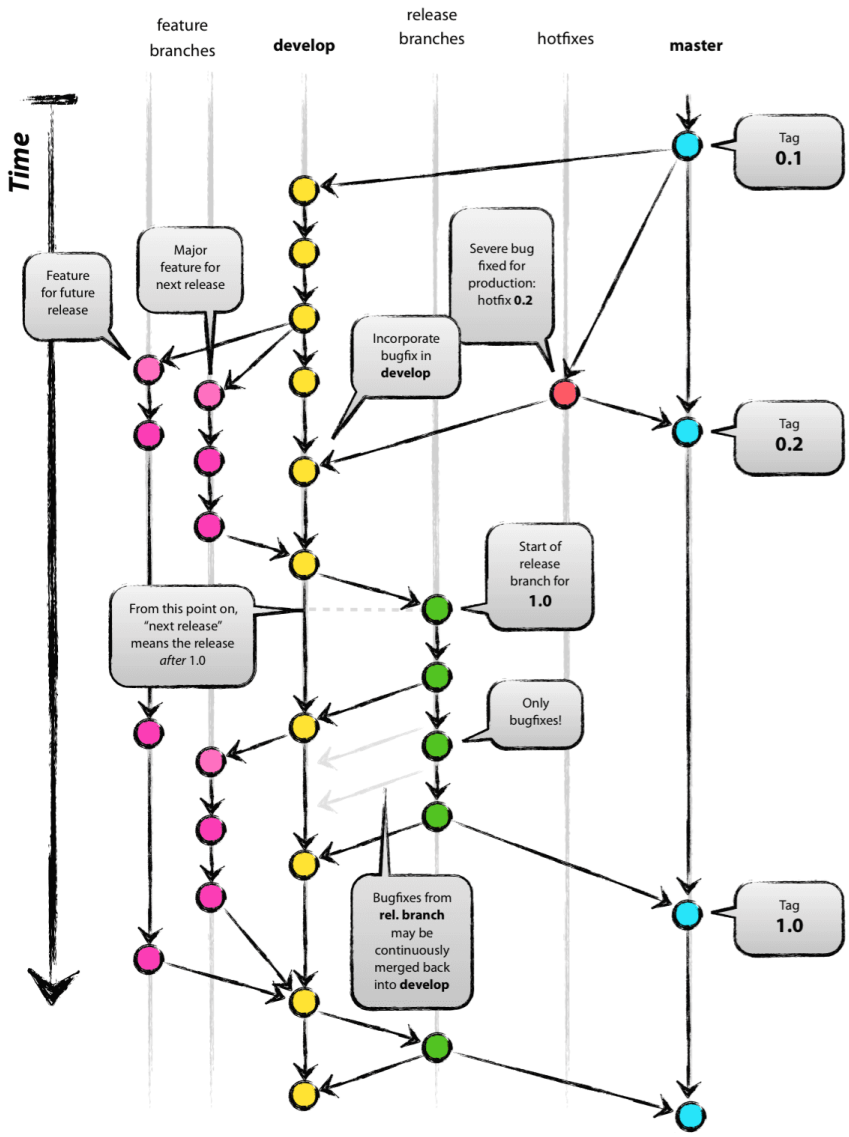
Un modelo de ramificación exitoso en git.

Miguel Piña

[2015-03-22 lun 11:32]

Contents

En este post presento el modelo de desarrollo que he ido introduciendo para todos mis proyectos (de trabajo y privados) desde hace un año, y el cual se ha mostrado realmente exitoso. He estado queriendo escribir sobre él desde hace un tiempo, pero nunca he encontrado el tiempo para hacerlo bien, hasta ahora. No voy a hablar de los detalles en los proyectos, si no de la estrategia de ramificación y gestión de lanzamientos.



Está enfocado alrededor de git como herramienta de versionado de todo nuestro código.

1 ¿Por qué GIT?

Para una discusión detallada acerca de los pros y contras de Git en comparación con otras herramientas de control de versiones, pueden consultar la

web. Hay toda una gran discusión sobre eso. Como desarrollador, prefiero Git sobre todas las otras herramientas que hay hoy en día. Git realmente ha cambiado la forma en que los desarrolladores piensan el mezclado y la ramificación de código. Desde el clásico CVS/Subversión del que vengo, siempre se han considerado de miedo ("¡Cuidado con los conflictos de mezclas, ellos pueden morderte!"), algo que pasa de vez en cuando.

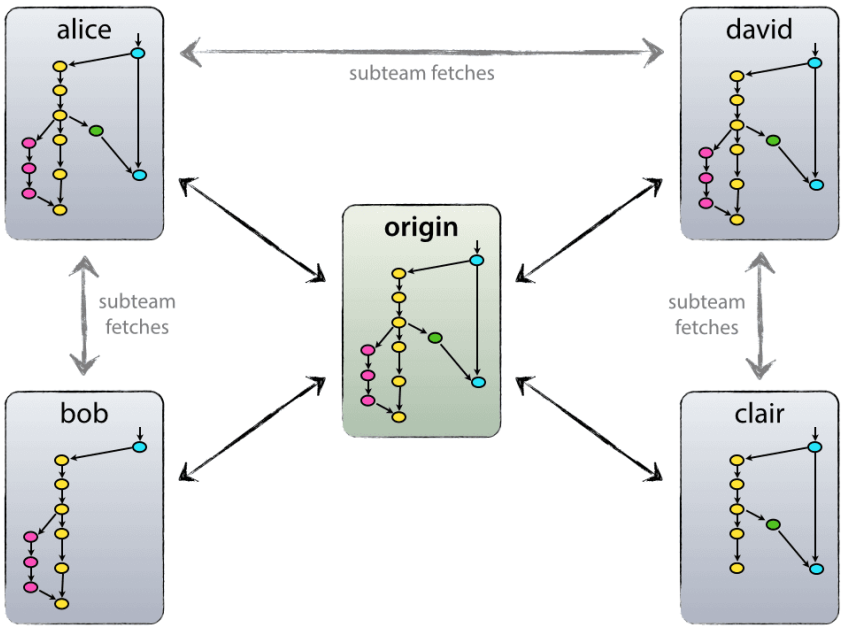
Pero con Git, esas acciones son extremadamente baratas y simples, las cuales son consideradas como parte del flujo de trabajo diario. Por ejemplo, en los libros de CVS/Subversión, la ramificación y el mezclado son discutidos en los capítulos finales (para usuarios avanzados), mientras que en cualquier libro de Git, esto es cubierto en el capítulo 3 (cosas básicas).

Como consecuencia de su simplicidad y naturaleza repetitiva, la ramificación y el mezclado ya no son cosas de las que hay que temer. Las herramientas de versionados se suponen que deben ayudar en la ramificación/mezclado sobre cualquier otra cosa.

Suficiente sobre las herramientas, vamos hacia el modelo de desarrollo. El modelo que voy a presentar aquí, es esencialmente un conjunto de procedimientos que todo miembro del equipo tiene que seguir en orden para tener un proceso de software administrado.

2 Descentralizado pero centralizado.

La configuración del repositorio que usamos y que trabaja bien con este modelo de ramificaciones, es ese con un repositorio central "verdadero". Note que este repositorio es sólo considerado como el central (ya que Git es un DVCS, no hay tal cosa como un repositorio central a nivel técnico). Nos vamos a referir a este repositorio como *origin*, ya que este nombre es familiar para todos los usuario de Git.



Cada desarrollar *pulls* y *pushes* a *origin*. Pero además de las relaciones *push-pull* centralizados, cada desarrollador puede también extraer los cambios de otro desarrollador. Por ejemplo, esto podría ser útil para que trabajen dos o más desarrolladores en una gran novedad antes de empujar el trabajo en curso con *origin* de manera prematura. En la figura anterior, hay equipos secundarios de Alice y Bob, Alice y David, y Clair y David.

Técnicamente, esto significa nada más que Alice ha definido un control, tomado desde el repositorio de bob, y viceversa.

3 Las ramas principales

En el núcleo, el modelo de desarrollo está inspirado por los modelos existentes que hay. El repositorio central mantiene dos ramas principales con vida infinita:

- Master
- Develop

La rama **master** en **origin** debería ser familiar para todo usuario de Git. Paralelo a **master**, existe otra rama llamada **develop**.

Nosotros consideramos a **origin/master** para ser la rama principal donde el código fuente de *HEAD* siempre apunta un estado *listo-para-producción*.

Nosotros consideramos a **origin/develop** para ser la rama principal donde el código fuente de *HEAD* siempre refleja el estado con los últimos cambios liberados para la siguiente liberación. Algunos podrían llamarla como la rama de "integración". Aquí es donde cualquier *nightly build* puede ser construida.

Cuando el código fuente en la rama **develop** alcanza un punto estable y está lista para ser liberada, todos los cambios deberían ser mezclados en **master** y entonces etiquetados con un número de versión. Más adelante vamos a discutir el cómo se realiza esto.

Por lo tanto, cada vez cuando los cambios son mezclados en **master**, esto implica que esta en producción por *definición*. Podemos ser muy estrictos con esto, por lo que, en teoría podemos tener un script de Git para construir nuestro proyecto de manera automática y poner en marcha nuestro software para nuestros servidores de producción cada vez que haya un commit en *master*.

4 Soportando ramificaciones

Además de las ramas principales *master* y *develop*, nuestro modelo de desarrollo usa una variedad de ramas de soporte para ayudar el desarrollo en paralelo entre los miembros del equipo, así como un fácil seguimiento de características, preparación para las liberaciones en producción y asistir rápidamente a resolver problemas con el sistema en producción. A diferencia de las ramas principales, esas ramas siempre tiene un tiempo de vida limitado, por lo que tendrán que ser eliminadas eventualmente.

Los diferentes tipos de ramas que tal vez usemos son:

- Ramas de características
- Ramas de liberación
- Ramas hotfix

Cada una de esas ramas tiene un propósito específico y están acotados por reglas estrictas, por ejemplo, de que ramas pueden ser originadas y cuales son las ramas con las que deben ser mezcladas. Vamos a revisarlas en un minuto.

De ninguna forma estas son ramas *especiales* desde una perspectiva técnica. Los tipos de ramas va estar categorizadas por la forma en que vamos a usarlas. Son por supuesto, las viejas y simples ramas de Git.

4.1 Ramas de características

Pueden ramificar desde:

- develop

Deben ser mezcaldas en:

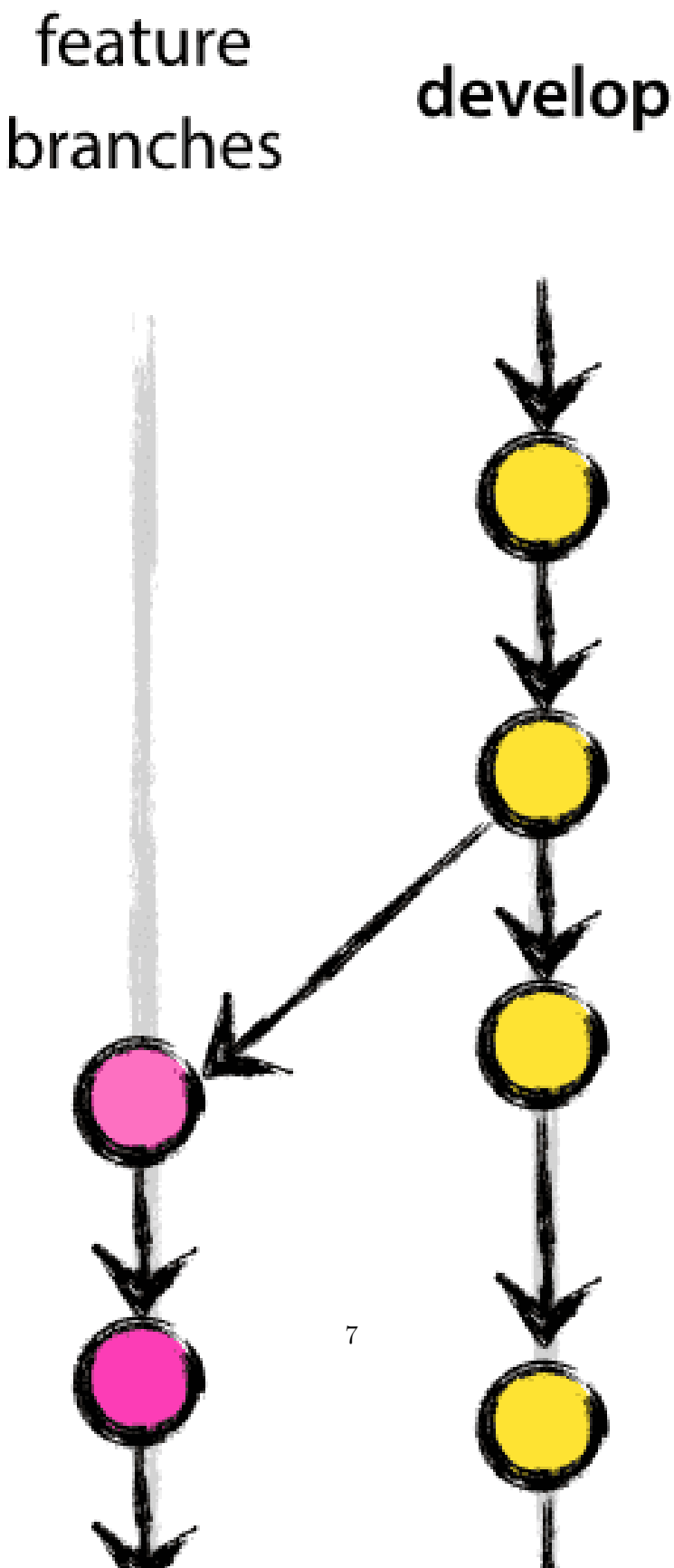
- Develop

Convención de nombres:

- Cualquiera, excepto: master, develop, release-* o hotfix-*

Las ramas de características (o algunas veces llamadas ramas puntuales) son usadas para desarrollar nuevas características a futuro o para una versión futura distante. Al iniciar el desarrollo de una característica, el release objetivo en la cuál esta característica va a ser incluida tal vez puede ser desconocida en ese punto. La esencia de una rama de característica es que existe siempre y cuando la función se encuentra en desarrollo, pero con el tiempo se mezcla en *develop* (añadida como una característica para la próxima liberación) o deshechada (en caso de ser un experimente descepcionante).

Las ramas de características normalmente viven en los repositorios de los desarrollador, no en *origin*.


feature
branches
develop

4.1.1 Creando una rama de características

Cuando iniciamos el trabajo en nueva característica, debemos ramificar desde la rama *develop*.

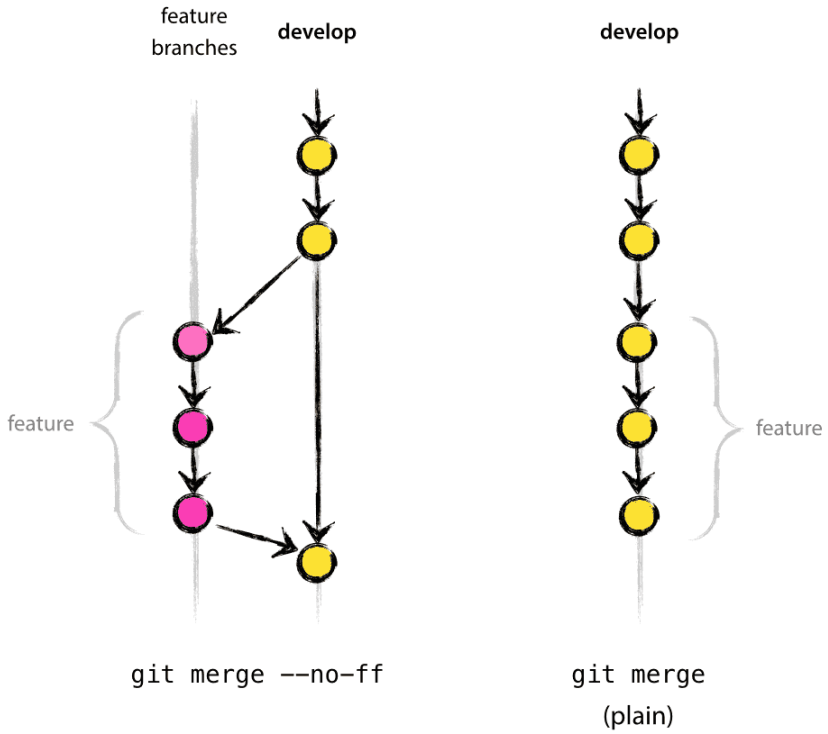
```
$ git checkout -b miCaracteristica develop
```

4.1.2 Incorporando una característica terminado en develop

Las características terminadas pueden ser mezcladas en la rama *develop* cuando van a ser añadidas de manera definitiva para la próxima liberación:

```
$ git checkout Develop
# Intercambio a la rama 'develop'
$ git merge --no-ff miCaracteristica
# Updating ea1b82a..05e9557
#(Summary of changes)
$ git branch -d myfeature
# Deleted branch myfeature (was 05e9557).
$ git push origin develop
```

El flag *–no-ff* causa que la mezcla siempre cree un nuevo objeto commit, incluso si la mezcla podría ser realizada con un fast-foward. Esto evita la perdida de información sobre la existencia histórica de una rama de características y agrupar todas las confirmaciones que agregan la función. Comparemos:



En el último caso, es imposible ver desde la historia de Git cuales fueron los commits que se hicieron para implementar una característica. Podrías tener que leer de manera manual todos los logs. Revertir toda una característica (conjunto de commits), es un dolor de cabeza en la última situación, mientras que es fácilmente hacerlo si hubieras usado el flag *–no-ff*.

Sí, vas a crear unos cuantos más commits (vacíos), pero la ganancia es mucho más grande que ese costo.

Por desgracia, no he encontrado una manera de hacer *–no-ff* el comportamiento predeterminado de *git merge*, pero realmente es lo que debería ser.

4.2 Ramas de lanzamiento

Pueden ser ramificadas desde:

- develop

Deben ser mezcladas en:

- develop y master

Convención de nombres:

- release-*

Las ramas de lanzamiento soportan la preparación de una nueva liberación en producción. Ellas permiten realizar cambios de último minuto. Así también realizar correcciones y preparar los metadatos para el lanzamiento (números de versión, fechas de construcción, etc). Haciendo todo este trabajo sobre una rama de lanzamiento, la rama de **develop** está lista para recibir características para el siguiente gran lanzamiento.

El principal momento para ramificar una nueba rama de lanzamiento desde **develop** es cuando el desarrollo refleja un estado deseado para una nueva liberación. Al menos todas las características son enviadas para el lanzamiento, construidas al ser mezcladas en **develop** en este punto. Todas las características enviadas a lanzamientos futuros no deben estar incluidas - deben esperar a su momento.

En el momento de iniciar una rama de lanzamiento cuando se le asigna un número de versión (no antes). Hasta ese momento, la rama **develop** refleja los cambios para la "próxima liberación", no está claro si ese próximo lanzamiento se convertirá en 0.3 o 1.0, hasta que se inicie la rama de lanzamiento.

4.2.1 Creando una rama de lanzamiento

La ramas de lanzamiento son creadas desde la rama **develop**. Por ejemplo, digamos que la versión 1.1.5 es la versión actual en producción y nosotros tenemos una gran liberación pronto. El estado de **develop** está listo para la "siguiente liberación" y nosotros tenemos que decidir que va ir en la versión 1.2 (en lugar de 1.1.6 o 2.0). Así nosotros ramificamos y damos la rama de lanzamiento el nombre que refleje el nuevo número de versión.

```
git checkout -b release-1.2 develop
# Creamos una nueva rama "release-1.2" y nos movemos a ella
./bump-version.sh 1.2
# Cambiamos los archivos indicado la versión 1.2
git commit -a -m "Cambio de versión a 1.2"
# [release-1.2 74d9424] Cambio de versión a 1.2
# 1 files changed, 1 insertions(+), 1 deletions(-)
```

Después de crear la nueva rama y movernos a ella, cambiamos el número de versión. Aquí **bump-version.sh** es un script imaginario que cambia algunos archivos en la copia de trabajo para reflejar la nueva versión (Esto

puede ser por supuesto un cambio manual, el punto es cambiar *algunos* archivos). Entonces, el número de versión es commiteado.

Esta nueva rama puede existir por un tiempo, hasta que la liberación sea terminada definitivamente. Durante ese tiempo, se reparan los errores que puedan existir en esa rama (en lugar de la rama de desarrollo). Añadir nuevas características grandes está estrictamente prohibido. Estas deben ser mezcladas en **develop** y esperar para el siguiente lanzamiento.

4.2.2 Terminando una rama de lanzamiento

Cuando el estado de una rama de lanzamiento está lista para ser un lanzamiento rela, algunas acciones deben ser realizadas. Primero, la rama de lanzamiento es mezclada en **master** (Debido a que cada commit en **master** es un nuevo lanzamiento por definición, recuerdas?). Después, ese commit en **master** debe ser etiquetado para tener una referencia futura. Finalmente, los cambios realizados en la rama de liberación deben ser mezclados de nuevo en **develop**.

Los primeros dos pasos en GIT:

```
git checkout master
# Switched to branch 'master'
git merge --no-ff release-1.2
# Merge made by recursive.
# (Summary of changes)
$ git tag -a 1.2
```

El lanzamiento está hecho, y etiquetado para futuras referencias.

Para mantener esos cambios realizados en la rama de liberación, debemos de mezclarlos de vuelta en **develop**. En GIT:

```
git checkout develop
# Switched to branch 'develop'
git merge --no-ff release-1.2
# Merge made by recursive.
# (Summary of changes)
```

Este paso podría tener algunos conflictos al mezclar (probablemente algunos, desde que cambios el número de versión). Si es así, los corregimos y hacemos commit.

Ahora debemos eliminar la rama de lanzamiento debido a que ya no la vamos a seguir usando.

```
git branch -d release-1.2
# Deleted branch release-1.2 (was ff452fe).
```

4.3 Ramas de correcciones

Pueden ser ramificadas desde:

- Master

Deben ser mezcladas en:

- develop y master

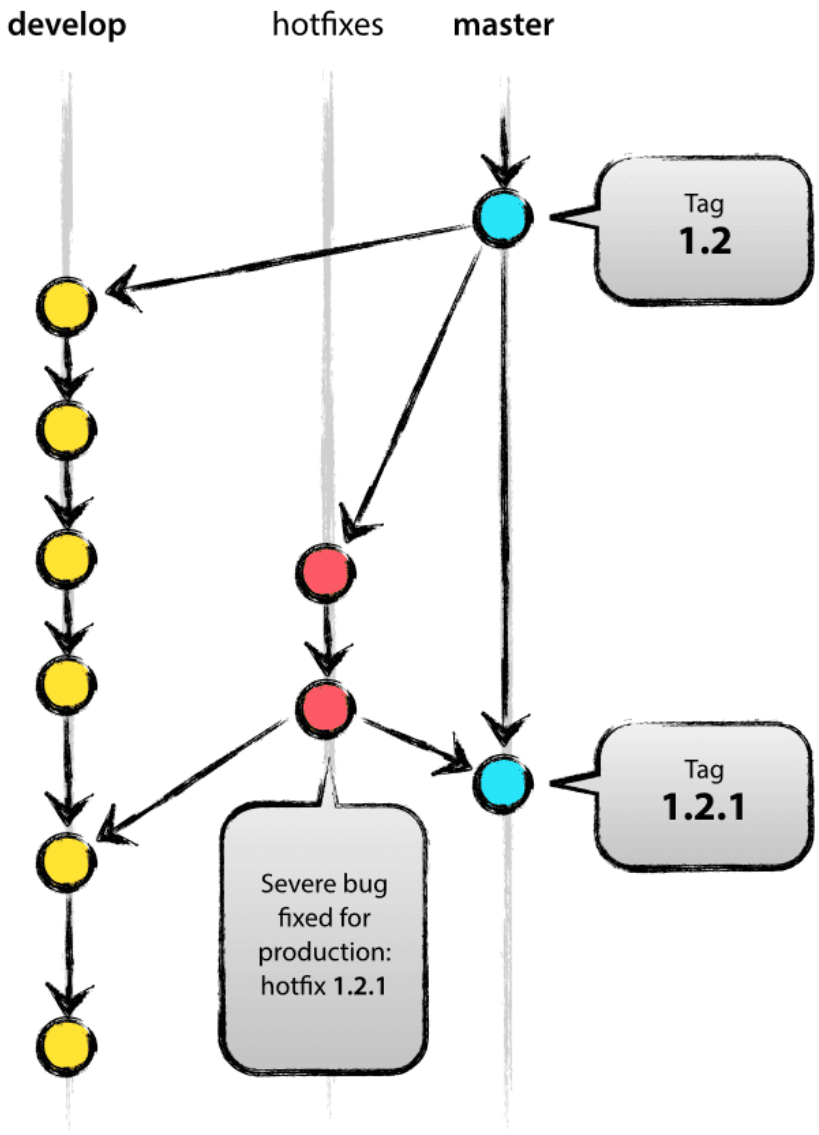
Convención de nombrado:

- hotfix-*

Las ramas de correción son muy parecidas a las ramas de lanzamiento en que ellas también se preparan para un nuevo lanzamiento en producción, aunque no sea planeado. Estas ramas nacen de la necesidad de cambiar un estado no deseado de una versión en producción. Cuando hay un error crítico en producción, esto debe ser resuelto inmediatamente, entonces se ramifica una rama de corrección a partir de la correspondiente etiqueta que marca la versión en producción.

Esencialmente el trabajo del equipo (sobre **develop**) puede continuar, mientras otra persona puede preparar una corrección rápida en producción.

4.3.1 Creando una rama de corrección



Las ramas de corrección son creadas a partir de la rama **master**. Por ejemplo, digamos que la versión 1.2 es la versión en producción actual y está causando problemas debido a un error. Pero los cambios en **develop** aún son inestables. Nosotros podemos ramificar una rama de corrección e iniciar a resolver el problema.

```
$ git checkout -b hotfix-1.2.1 master
# Switched to a new branch "hotfix-1.2.1"
$ ./bump-version.sh 1.2.1
# Files modified successfully, version bumped to 1.2.1.
$ git commit -a -m "Bumped version number to 1.2.1"
# [hotfix-1.2.1 41e61bb] Bumped version number to 1.2.1
# 1 files changed, 1 insertions(+), 1 deletions(-)
```

¡No olvides realizar el cambio de versión después de ramificar!

Entonces, corregimos el error y comiteamos el error en uno o más commits separados.

```
git commit -m "Fixed severe production problem"
# [hotfix-1.2.1 abbe5d6] Fixed severe production problem
# 5 files changed, 32 insertions(+), 17 deletions(-)
```

4.3.2 Terminando una rama de corrección

Cuando terminemos, la corrección del error necesita ser mezclada de nuevo en **master**, pero también necesitamos mezclarla de vuelta en **develop**, de forma que podamos asegurar que la corrección sea incluida en la siguiente versión también. Esto es completamente similar a como las ramas de lanzamiento son terminadas.

Primero, actualizamos master y la etiqueta de la liberación:

```
git checkout master
# Switched to branch 'master'
git merge --no-ff hotfix-1.2.1
# Merge made by recursive.
# (Summary of changes)
git tag -a 1.2.1
```

Lo siguiente es incluir la corrección en **develop**:

```
git checkout develop
# Switched to branch 'develop'
git merge --no-ff hotfix-1.2.1
# Merge made by recursive.
# (Summary of changes)
```

La única excepción a la regla es que, **cuando una rama de lanzamiento exista, la corrección necesita ser mezclada en al rama de liberación** en lugar de la de **develop**. Mezclando la corrección en la rama de lanzamiento, eventualmente va a ser mezclada en **develop** también. (Si el trabajo en **develop** requiere mezclar esta corrección y no pueden esperar hasta que la rama de lanzamiento termine, puedes mezclar la corrección seguramente en **develop** también).

Finalmente, eliminamos la rama de corrección:

```
git branch -d hotfix-1.2.1
# Deleted branch hotfix-1.2.1 (was abbe5d6).
```

Comentario: Este documento es una traducción de "A succesful Git branching model".