INTRODUCCIÓN A GIT Y GITHUB - DÍA 1

SERGIO GÓMEZ BACHILLER

Esta página se ha dejado vacía a propósito

Índice de contenidos

Esta página se ha dejado vacía a propósito

Capítulo 1

Sistemas de control de versiones

1.1 Definición, clasificación y funcionamiento

Se llama control de versiones a la gestión de los diversos cambios que se realizan sobre los elementos de algún producto o una configuración del mismo. Una versión, revisión o edición de un producto, es el estado en el que se encuentra dicho producto en un momento dado de su desarrollo o modificación. Aunque un sistema de control de versiones puede realizarse de forma manual, es muy aconsejable disponer de herramientas que faciliten esta gestión dando lugar a los llamados sistemas de control de versiones o SVC (del inglés System Version Control).

Estos sistemas facilitan la administración de las distintas versiones de cada producto desarrollado, así como las posibles especializaciones realizadas (por ejemplo, para algún cliente específico). Ejemplos de este tipo de herramientas son entre otros: CVS, Subversion, SourceSafe, ClearCase, Darcs, Bazaar , Plastic SCM, Git, Mercurial, Perforce.

1.1.1 Terminología

Repositorio ("repository")

El repositorio es el lugar en el que se almacenan los datos actualizados e históricos de cambios.

Revisión ("revision")

Una revisión es una versión determinada de la información que se gestiona. Hay sistemas que identifican las revisiones con un contador (Ej. subversion). Hay otros sistemas que identifican las revisiones mediante un código de detección de modificaciones (Ej. git usa SHA1).

Etiqueta ("tag")
Los tags permiten identificar de forma fácil revisiones importantes en el proyecto. Por ejemplo se suelen usar tags para identificar el contenido de las versiones publicadas del proyecto.

Rama ("branch")
Un conjunto de archivos puede ser ramificado o bifurcado en un punto en el tiempo de manera que, a partir de ese momento, dos copias de esos archivos se pueden desarrollar a velocidades diferentes o en formas diferentes de forma independiente el uno del otro.

Cambio ("change")
Un cambio (o diff, o delta) representa una modificación específica de un documento bajo el control de versiones. La granularidad de la modificación que es considerada como un cambio varía entre los sistemas de control de versiones.

Desplegar ("checkout")
Es crear una copia de trabajo local desde el repositorio. Un usuario puede especificar una revisión en concreto u obtener la última. El término 'checkout' también se puede utilizar como un sustantivo para describir la copia de trabajo.

Confirmar ("commit")
Confirmar es escribir o mezclar los cambios realizados en la copia de trabajo del repositorio. Los términos 'commit' y 'checkin' también se pueden utilizar como sustantivos para describir la nueva revisión que se crea como resultado de confirmar.

Conflicto ("conflict")
Un conflicto se produce cuando diferentes partes realizan cambios en el mismo documento, y el sistema es incapaz de conciliar los cambios. Un usuario debe resolver el conflicto mediante la integración de los cambios, o mediante la selección de un cambio en favor del otro.

Cabeza ("head")
También a veces se llama tip (punta) y se refiere a la última confirmación, ya sea en el tronco ('trunk') o en una rama ('branch'). El tronco y cada rama tienen su propia cabeza, aunque HEAD se utiliza a veces libremente para referirse al tronco.

Tronco ("trunk")
La única línea de desarrollo que no es una rama (a veces también llamada línea base, línea principal o máster).

Fusionar, integrar, mezclar ("merge")

Una fusión o integración es una operación en la que se aplican dos tipos de cambios en un archivo o conjunto de archivos. Algunos escenarios de ejemplo son los siguientes:

- Un usuario, trabajando en un conjunto de archivos, actualiza o sincroniza su copia de trabajo con los cambios realizados y confirmados, por otros usuarios, en el repositorio.
- Un usuario intenta confirmar archivos que han sido actualizado por otros usuarios desde el último despliegue ('checkout'), y el software de control de versiones integra automáticamente los archivos (por lo general, después de preguntarle al usuario si se debe proceder con la integración automática, y en algunos casos sólo se hace si la fusión puede ser clara y razonablemente resuelta).
- Un conjunto de archivos se bifurca, un problema que existía antes de la ramificación se trabaja en una nueva rama, y la solución se combina luego en la otra rama.
- Se crea una rama, el código de los archivos es independiente editado, y la rama actualizada se incorpora más tarde en un único tronco unificado.

1.1.2 Clasificación

Podemos clasificar los sistemas de control de versiones atendiendo a la arquitectura utilizada para el almacenamiento del código:

Locales

Los cambios son guardados localmente y no se comparten con nadie. Esta arquitectura es la antecesora de las dos siguientes.

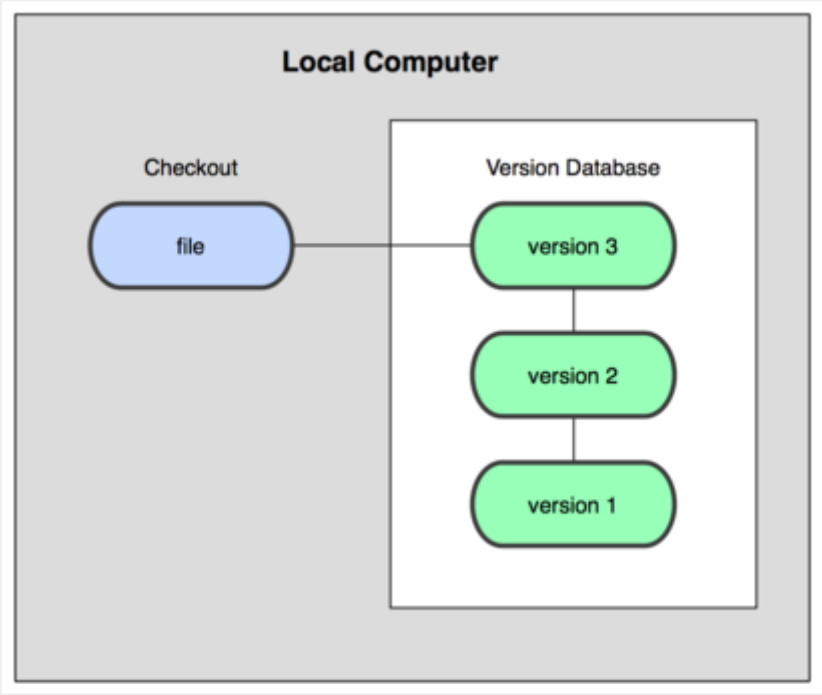

Figura 1.1 Sistema de control de versiones local

Centralizados

Existe un repositorio centralizado de todo el código, del cual es responsable un único usuario (o conjunto de ellos). Se facilitan las tareas administrativas a cambio de reducir flexibilidad, pues todas las decisiones fuertes (como crear una nueva rama) necesitan la aprobación del responsable. Algunos ejemplos son CVS y Subversion.

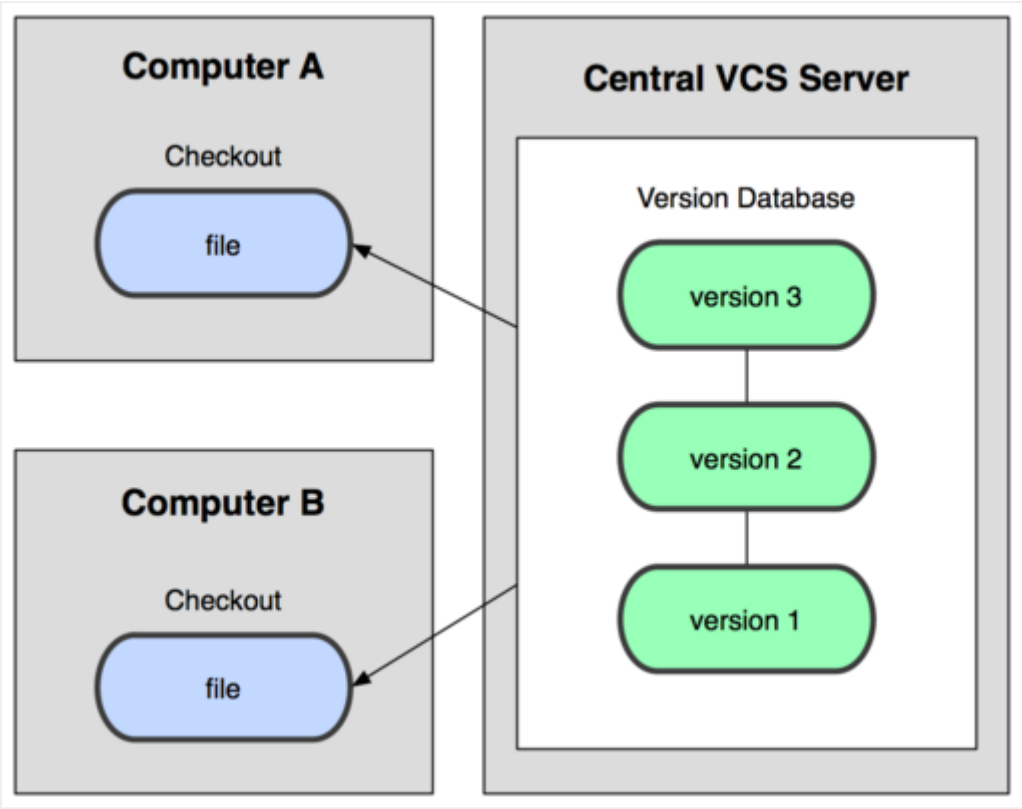

Figura 1.2 Sistema de control de versiones centralizado

Distribuidos

Cada usuario tiene su propio repositorio. Los distintos repositorios pueden intercambiar y mezclar revisiones entre ellos. Es frecuente el uso de un repositorio, que está normalmente disponible, que sirve de punto de sincronización de los distintos repositorios locales. Ejemplos: Git y Mercurial.

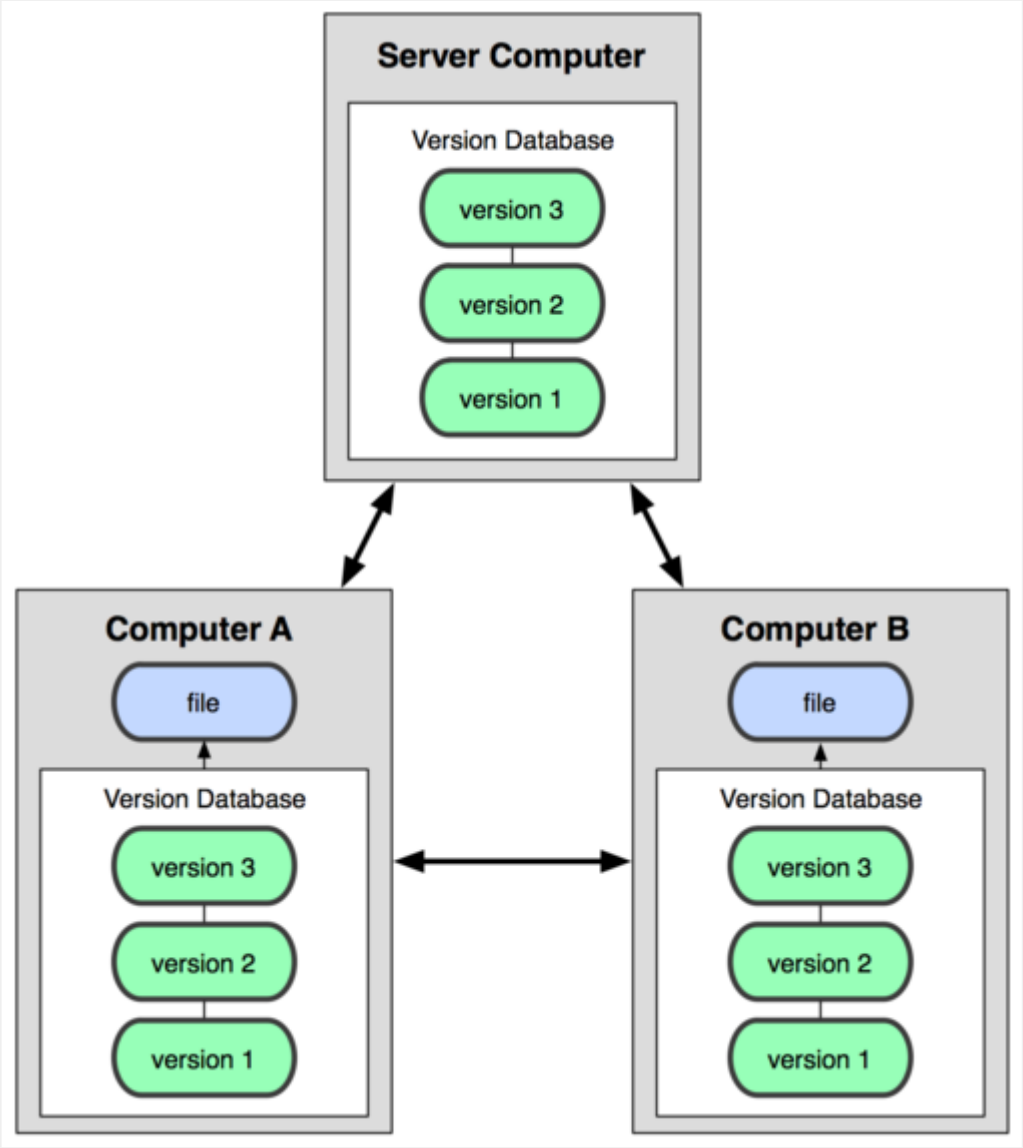

Figura 1.3 Sistema de control de versiones distribuido

Ventajas de sistemas distribuidos

- No es necesario estar conectado para guardar cambios.
- Posibilidad de continuar trabajando si el repositorio remoto no está accesible.
- El repositorio central está más libre de ramas de pruebas.

- Se necesitan menos recursos para el repositorio remoto.
- Más flexibles al permitir gestionar cada repositorio personal como se quiera.

1.2 Introducción a git

Git es un sistema de control de versiones distribuido que se diferencia del resto en el modo en que modela sus datos. La mayoría de los demás sistemas almacenan la información como una lista de cambios en los archivos, mientras que Git modela sus datos más como un conjunto de instantáneas de un mini sistema de archivos.

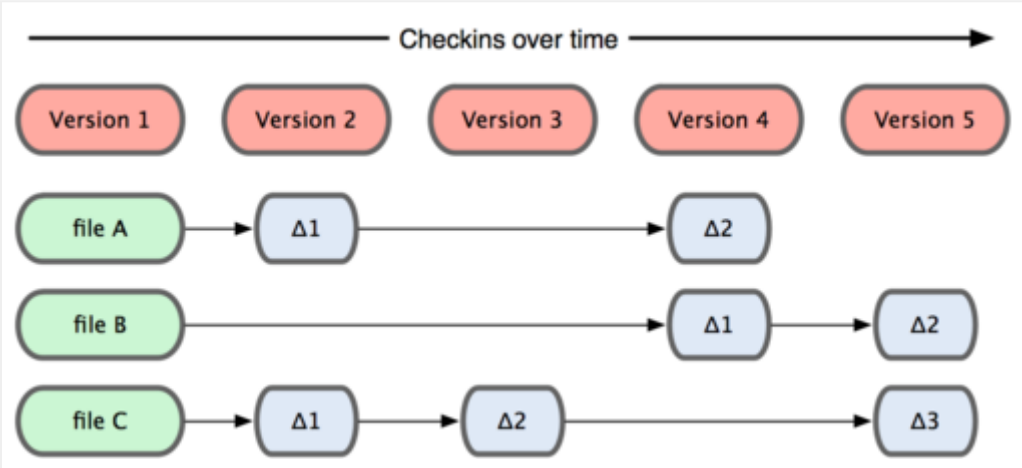

Figura 1.4 Modelo de datos de los sistemas distribuidos tradicionales

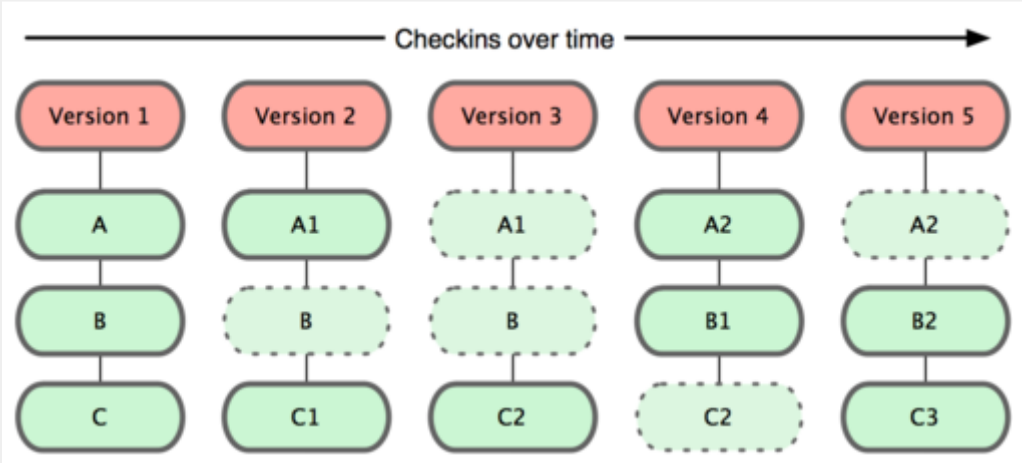

Figura 1.5 Modelo de datos de Git

1.2.1 Los tres estados

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).

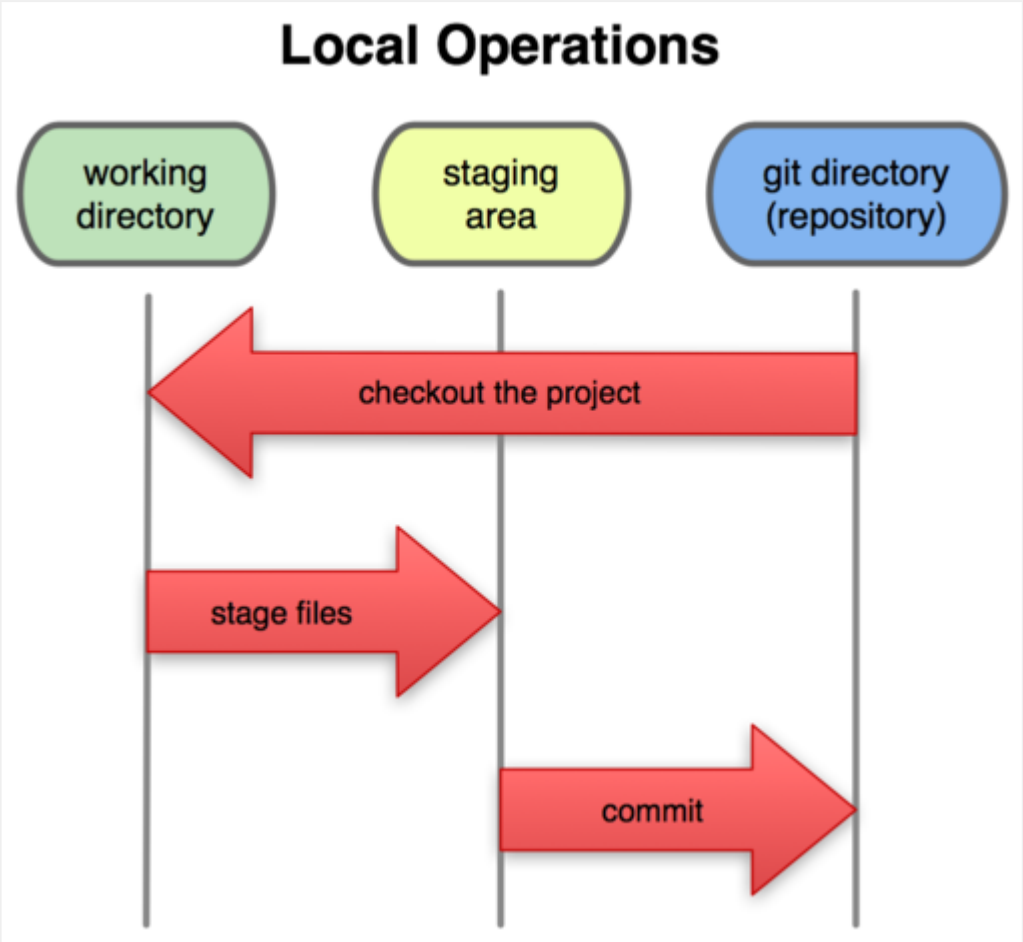

Figura 1.6 Directorio de trabajo, área de preparación, y directorio de Git

1.2.2 Flujos de trabajo distribuidos con git

Hemos visto en qué consiste un entorno de control de versiones distribuido, pero más allá de la simple definición, existe más de una manera de gestionar los repositorios. Estos son los flujos de trabajo más comunes en Git.

Flujo de trabajo centralizado

Existe un único repositorio o punto central que guarda el código y todo el mundo sincroniza su trabajo con él. Si dos desarrolladores clonan desde el punto central, y ambos hacen cambios; tan solo el primero de ellos en enviar sus cambios de vuelta lo podrá hacer limpiamente. El segundo desarrollador deberá fusionar previamente su trabajo con el del primero, antes de enviarlo, para evitar el sobreescribir los cambios del primero

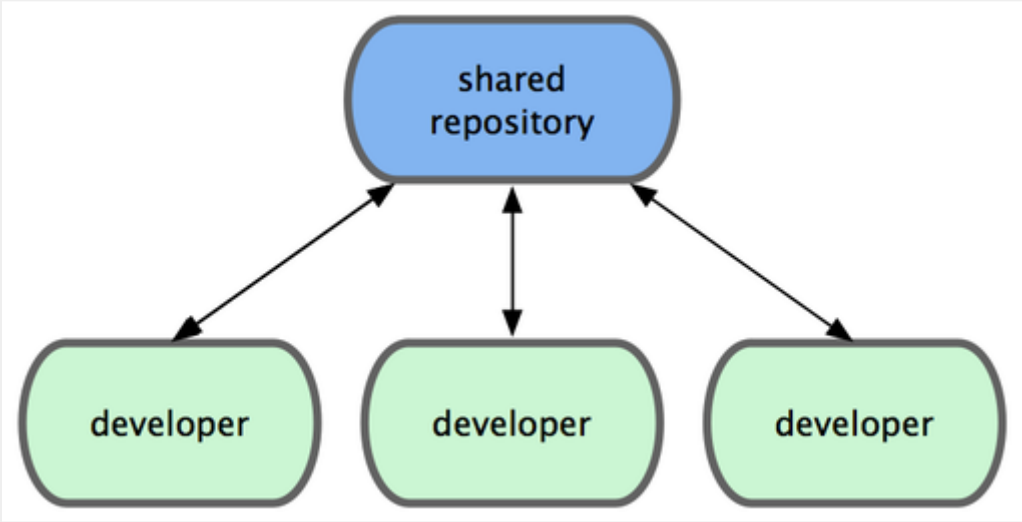

Figura 1.7 Flujo de trabajo centralizado

Flujo de trabajo del Gestor-de-Integraciones

Al permitir multiples repositorios remotos, en Git es posible tener un flujo de trabajo donde cada desarrollador tenga acceso de escritura a su propio repositorio público y acceso de lectura a los repositorios de todos los demás. Habitualmente, este escenario suele incluir un repositorio canónico, representante "oficial" del proyecto. Este modelo se puso muy de moda a raíz de la forja GitHub.

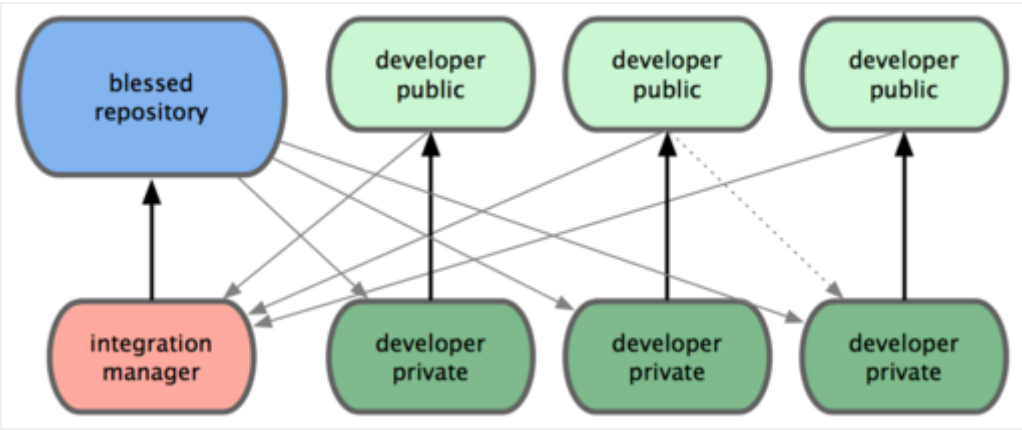

Figura 1.8 Flujo de trabajo del Gestor-de-Integraciones

Flujo de trabajo con Dictador y Tenientes

Es una variante del flujo de trabajo con multiples repositorios. Se utiliza generalmente en proyectos muy grandes, con cientos de colaboradores. Un ejemplo muy conocido es el del kernel de Linux. Unos gestores de integración se encargan de partes concretas del repositorio; y se denominan tenientes. Todos los tenientes rinden cuentas a un gestor de integración; conocido como el dictador benevolente. El repositorio del dictador benevolente es el repositorio de referencia, del que recuperan (pull) todos los colaboradores.

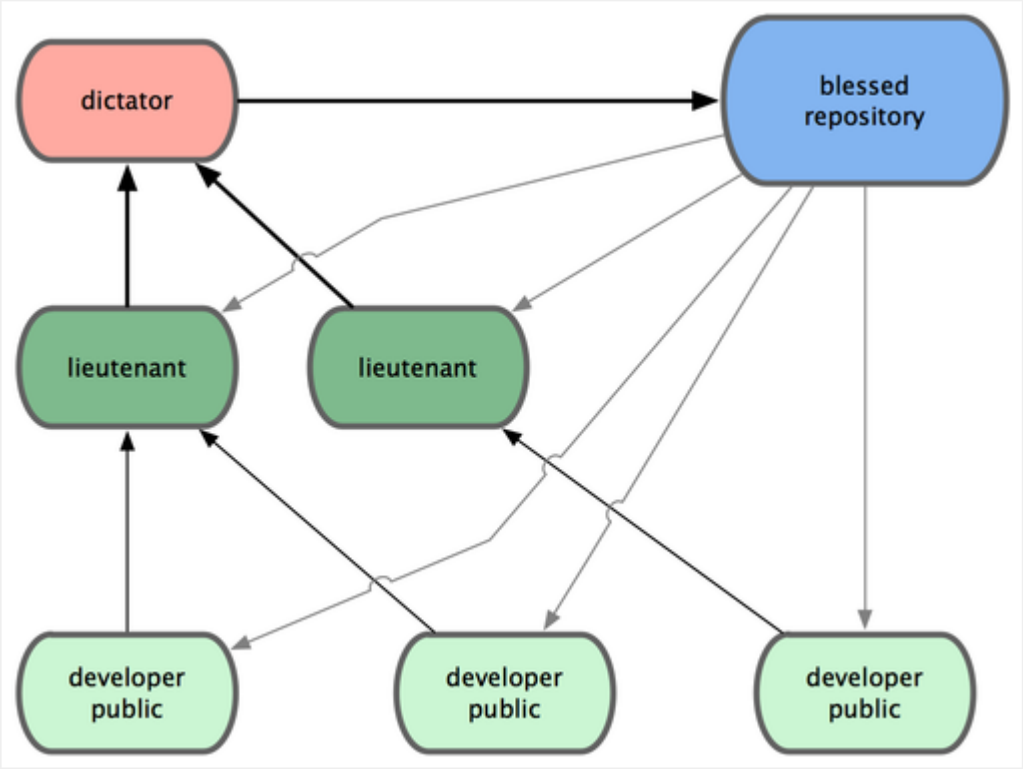

Figura 1.9 Flujo de trabajo con Dictador y Tenientes

Esta página se ha dejado vacía a propósito

Capítulo 2

Aspectos básicos de Git

2.1 Instalación

La instalación del cliente Git no supone mayor problema. Desde la web oficial de Git (https://git-scm.com/) puedes obtener las instrucciones para instalarlo en las diferentes distribuciones de Linux, así mismo podrás encontrar el instalador para Windows. En OSX ya viene instalado de serie, aunque desde la web oficial podrás bajar la última versión.

2.2 Configuración

2.2.1 Tu identidad

Lo primero que deberías hacer cuando instalas Git es establecer tu nombre de usuario y dirección de correo electrónico. Esto es importante porque las confirmaciones de cambios (commits) en 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
```

Si la versión que tienes instalada es inferior a la 2.0 (cosa poco probable si lo acabas de instalar) se recomienda configurar el siguiente parámetro:

```
$ git config --global push.default simple
```

2.2.2 Bash Completion

Bash completion es una utilidad que permite a bash completar órdenes y parámetros. Por defecto suele venir desactivada en Ubuntu y es necesario modificar el archivo `$HOME/.bashrc` para poder activarla. Simplemente hay que descomentar las líneas que lo activan,

En OSX no está disponible de serie. Para disponer de bash_completion previamente deberías tener alguna herramienta para instalación de paquetes estilo `homebrew` o `macports`. Se recomienda que instales homebrew (http://brew.sh/) puesto que ofrece una interfaz similar a la que trae `apt-get` o `yum` en Linux.

2.2.3 Tu clave pública/privada

Muchos servidores Git utilizan la autentificación a través de claves públicas SSH. Y, para ello, cada usuario del sistema ha de generarse una, si es que no la tiene ya. El proceso para hacerlo es similar en casi cualquier sistema operativo. Ante todo, asegurarte que no tengas ya una clave. (comprueba que el directorio `$HOME/usuario/.ssh` no tiene un archivo id_dsa.pub o id_rsa.pub).

Para crear una nueva clave usamos la siguiente orden:

```
$ ssh-keygen -t rsa -C "correo@servidor.es"
```

El parámetro "-C" no es obligatorio, solo sirve para identificar la clave por si vamos a crear más de una.

Capítulo 3

Uso básico de Git

3.1 Crear un proyecto

3.1.1 Crear un programa "Hola Mundo"

Creamos un directorio donde colocar el código

```
$ mkdir curso-de-git
$ cd curso-de-git
```

Creamos un fichero `hola.php` que muestre Hola Mundo.

```
<?php
echo "Hola Mundo\n";
```

3.1.2 Crear el repositorio

Para crear un nuevo repositorio se usa la orden `git init`

```
$ git init
Initialized empty Git repository in /home/cc0gobas/git/curso-de-git/.git/
```

3.1.3 Añadir la aplicación

Vamos a almacenar el archivo que hemos creado en el repositorio para poder trabajar, después explicaremos para qué sirve cada orden.

```
$ git add hola.php
$ git commit -m "Creación del proyecto"
[master (root-commit) e19f2c1] Creación del proyecto
```

```
 1 file changed, 2 insertions(+)
 create mode 100644 hola.php
```

3.1.4 Comprobar el estado del repositorio

Con la orden `git status` podemos ver en qué estado se encuentran los archivos de nuestro repositorio.

```
$ git status
# On branch master
nothing to commit (working directory clean)
```

Si modificamos el archivo `hola.php`:

```
<?php
@print "Hola {$argv[1]}\n";
```

Y volvemos a comprobar el estado del repositorio:

```
$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working direct
ory)
#
#   modified:   hola.php
#
no changes added to commit (use "git add" and/or "git commit -a")
```

3.1.5 Añadir cambios

Con la orden `git add` indicamos a git que prepare los cambios para que sean almacenados.

```
$ git add hola.php
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   modified:   hola.php
#
```

3.1.6 Confirmar los cambios

Con la orden `git commit` confirmamos los cambios definitivamente, lo que hace que se guarden permanentemente en nuestro repositorio.

```
$ git commit -m "Parametrización del programa"
[master efc252e] Parametrización del programa
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git status
# On branch master
nothing to commit (working directory clean)
```

3.1.7 Funcionamiento de git: se guardan cambios, no ficheros.

Modificamos nuestra aplicación para que soporte un parámetro por defecto y añadimos los cambios.

```
<?php
$nombre = isset($argv[1]) ? $argv[1] : "Mundo";
@print "Hola, {$nombre}\n";
```

```
git add hola.php
```

Volvemos a modificar el programa para indicar con un comentario lo que hemos hecho.

```
<?php
// El nombre por defecto es Mundo
$nombre = isset($argv[1]) ? $argv[1] : "Mundo";
@print "Hola, {$nombre}\n";
```

Y vemos el estado en el que está el repositorio

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   modified:   hola.php
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working direct
ory)
#
#   modified:   hola.php
#
```

Almacenamos los cambios por separado:

```
$ git commit -m "Se añade un parámetro por defecto"
[master 3283e0d] Se añade un parámetro por defecto
 1 file changed, 2 insertions(+), 1 deletion(-)
$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working direct
ory)
#
#   modified:   hola.php
#
no changes added to commit (use "git add" and/or "git commit -a")
$ git add .
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   modified:   hola.php
#
$ git commit -m "Se añade un comentario al cambio del valor por defecto"
[master fd4da94] Se añade un comentario al cambio del valor por defecto
 1 file changed, 1 insertion(+)
```

El valor "." despues de `git add` indica que se añadan todos los archivos que hayan cambiado.

3.2 Trabajando con el historial

3.2.1 Observando los cambios

Con la orden `git log` podemos ver todos los cambios que hemos hecho:

```
$ git log
commit fd4da946326fbe8b24e89282ad25a71721bf40f6
Author: Sergio Gómez <sergio@uco.es>
Date:   Sun Jun 16 12:51:01 2013 +0200

    Se añade un comentario al cambio del valor por defecto

commit 3283e0d306c8d42d55ffcb64e456f10510df8177
Author: Sergio Gómez <sergio@uco.es>
Date:   Sun Jun 16 12:50:00 2013 +0200
```

```
    Se añade un parámetro por defecto

commit efc252e11939351505a426a6e1aa5bb7dc1dd7c0
Author: Sergio Gómez <sergio@uco.es>
Date:   Sun Jun 16 12:13:26 2013 +0200

    Parametrización del programa

commit e19f2c1701069d9d1159e9ee21acaa1bbc47d264
Author: Sergio Gómez <sergio@uco.es>
Date:   Sun Jun 16 11:55:23 2013 +0200

    Creación del proyecto
```

También es posible ver versiones abreviadas o limitadas, dependiendo de los parámetros:

```
$ git log --oneline
fd4da94 Se añade un comentario al cambio del valor por defecto
3283e0d Se añade un parámetro por defecto
efc252e Parametrización del programa
e19f2c1 Creación del proyecto
git log --oneline --max-count=2
git log --oneline --since='5 minutes ago'
git log --oneline --until='5 minutes ago'
git log --oneline --author=sergio
git log --oneline --all
```

Una versión muy útil de `git log` es la siguiente, pues nos permite ver en que lugares está master y HEAD, entre otras cosas:

```
$ git log --pretty=format:'%h %ad | %s%d [%an]' --graph --date=short
* fd4da94 2013-06-16 | Se añade un comentario al cambio del valor por def
ecto (HEAD, master) [Sergio Gómez]
* 3283e0d 2013-06-16 | Se añade un parámetro por defecto [Sergio Gómez]
* efc252e 2013-06-16 | Parametrización del programa [Sergio Gómez]
* e19f2c1 2013-06-16 | Creación del proyecto [Sergio Gómez]
```

3.2.2 Crear alias

Como estas órdenes son demasiado largas, Git nos permite crear alias para crear nuevas órdenes parametrizadas. Para ello editaremos un archivo llamado `.gitconfig` que está en nuestro `$HOME` y le añadiremos estas líneas al final:

```
[alias]
  hist = log --pretty=format:'%h %ad | %s%d [%an]' --graph --date=short
```

3.2.3 Recuperando versiones anteriores

Cada cambio es etiquetado por un hash, para poder regresar a ese momento del estado del proyecto se usa la orden `git checkout`.

```
$ git checkout e19f2c1
Note: checking out 'e19f2c1'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Exampl
e:

  git checkout -b new_branch_name

HEAD is now at e19f2c1... Creación del proyecto
$ cat hola.php
<?php
echo "Hello, World\n";
```

El aviso que nos sale nos indica que estamos en un estado donde no trabajamos en ninguna rama concreta. Eso significa que los cambios que hagamos podrían "perderse" porque si no son guardados en una nueva rama, en principio no podríamos volver a recuperarlos. Hay que pensar que Git es como un árbol donde un nodo tiene información de su nodo padre, no de sus nodos hijos, con lo que siempre necesitaríamos información de dónde se encuentran los nodos finales o de otra manera no podríamos acceder a ellos.

3.2.4 Volver a la última versión de la rama master.

Usamos `git checkout` indicando el nombre de la rama:

```
$ git checkout master
Previous HEAD position was e19f2c1... Creación del proyecto
```

3.2.5 Etiquetando versiones

Para poder recuperar versiones concretas en la historia del repositorio, podemos etiquetarlas, lo cual es más facil que usar un hash. Para eso usaremos la orden `git tag`.

```
$ git tag v1
```

Ahora vamos a etiquetar la versión inmediatamente anterior como v1-beta. Para ello podemos usar los modificadores ^ o ~ que nos llevarán a un ancestro determinado. Las siguientes dos órdenes son equivalentes:

```
$ git checkout v1^
$ git checkout v1~1
$ git tag v1-beta
```

Si ejecutamos la orden sin parámetros nos mostrará todas las etiquetas existentes.

```
$ git tag
v1
v1-beta
```

Y para verlas en el historial:

```
$ git hist master --all
* fd4da94 2013-06-16 | Se añade un comentario al cambio del valor por def
ecto (tag: v1, master) [Sergio Gómez]
* 3283e0d 2013-06-16 | Se añade un parámetro por defecto (HEAD, tag: v1-b
eta) [Sergio Gómez]
* efc252e 2013-06-16 | Parametrización del programa [Sergio Gómez]
* e19f2c1 2013-06-16 | Creación del proyecto [Sergio Gómez]
```

3.2.6 Borrar etiquetas

Para borrar etiquetas:

```
git tag -d nombre_etiqueta
```

3.2.7 Visualizar cambios

Para ver los cambios que se han realizado en el código usamos la orden `git diff`. La orden sin especificar nada más, mostrará los cambios que no han sido añadidos aún, es decir, todos los cambios que se han hecho antes de usar la orden `git add`. Después se puede indicar un parámetro y dará los cambios entre la versión indicada y el estado actual. O para comparar dos versiones entre sí, se indica la más antigua y la más nueva. Ejemplo:

```
$ git diff v1-beta v1
diff --git a/hola.php b/hola.php
index a31e01f..25a35c0 100644
--- a/hola.php
+++ b/hola.php
@@ -1,3 +1,4 @@
 <?php
+// El nombre por defecto es Mundo
```

```
$nombre = isset($argv[1]) ? $argv[1] : "Mundo";
@print "Hola, {$nombre}\n";
```

Capítulo 4

Uso avanzado de Git

4.1 Deshacer cambios

4.1.1 Deshaciendo cambios antes de la fase de staging.

Volvemos a la rama máster y vamos a modificar el comentario que pusimos:

```
$ git checkout master
Previous HEAD position was 3283e0d... Se añade un parámetro por defecto
Switched to branch 'master'
$ # Modificamos hola.php
$ cat hola.php
<?php
// Este comentario está mal y hay que borrarlo
$nombre = isset($argv[1]) ? $argv[1] : "Mundo";
@print "Hola, {$nombre}\n";
$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working direct
ory)
#
#   modified:   hola.php
#
no changes added to commit (use "git add" and/or "git commit -a")
```

El mismo Git nos indica que debemos hacer para añadir los cambios o para deshacerlos:

```
$ git checkout hola.php
$ git status
# On branch master
nothing to commit, working directory clean
$ cat hola.php
<?php
// El nombre por defecto es Mundo
$nombre = isset($argv[1]) ? $argv[1] : "Mundo";
@print "Hola, {$nombre}\n";
```

4.1.2 Deshaciendo cambios antes del commit

Vamos a hacer lo mismo que la vez anterior, pero esta vez sí añadiremos el cambio al stag (sin hacer commit).

```
$ # Modificamos hola.php
$ cat hola.php
<?php
// Este comentario está mal y hay que borrarlo
$nombre = isset($argv[1]) ? $argv[1] : "Mundo";
@print "Hola, {$nombre}\n";
$ git add hola.php
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   modified:   hola.php
#
```

De nuevo, Git nos indica qué debemos hacer para deshacer el cambio almacenado en el stag:

```
$ git reset HEAD hola.php
Unstaged changes after reset:
M   hola.php
$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working direct
ory)
#
#   modified:   hola.php
#
```

```
no changes added to commit (use "git add" and/or "git commit -a")
$ git checkout hola.php
```

Y ya tenemos nuestro repositorio limpio otra vez.

4.1.3 Deshaciendo commits no deseados.

Si a pesar de todo hemos hecho un commit y nos hemos equivocado, podemos deshacerlo con la orden `git revert`. Modificamos otra vez el archivo como antes pero ahora sí hacemos commit:

```
$ # Modificamos hola.php
$ cat hola.php
<?php
// Este comentario está mal y hay que borrarlo
$nombre = isset($argv[1]) ? $argv[1] : "Mundo";
@print "Hola, {$nombre}\n";
$ git add hola.php
$ git commit -m "Ups... este commit está mal."
master 5a5d067] Ups... este commit está mal
 1 file changed, 1 insertion(+), 1 deletion(-)
```

4.1.4 Crear un "reverting commit"

```
$ git revert HEAD --no-edit
[master 817407b] Revert "Ups... este commit está mal"
1 file changed, 1 insertion(+), 1 deletion(-)
$ git hist
* 817407b 2013-06-16 | Revert "Ups... este commit está mal" (HEAD, maste
r) [Sergio Gómez]
* 5a5d067 2013-06-16 | Ups... este commit está mal [Sergio Gómez]
* fd4da94 2013-06-16 | Se añade un comentario al cambio del valor por def
ecto (tag: v1) [Sergio Gómez]
* 3283e0d 2013-06-16 | Se añade un parámetro por defecto (tag: v1-beta)
[Sergio Gómez]
* efc252e 2013-06-16 | Parametrización del programa [Sergio Gómez]
* e19f2c1 2013-06-16 | Creación del proyecto [Sergio Gómez]
```

4.1.5 Borrar commits de una rama

El anterior apartado revierte un commit, pero deja huella en el historial de cambios. Para hacer que no aparezca hay que usar la orden `git reset`.

```
$ git reset --hard v1
HEAD is now at fd4da94 Se añade un comentario al cambio del valor por def
ecto
$ git hist
```

```
* fd4da94 2013-06-16 | Se añade un comentario al cambio del valor por def
ecto (HEAD, tag: v1, master) [Sergio Góme
* 3283e0d 2013-06-16 | Se añade un parámetro por defecto (tag: v1-beta)
[Sergio Gómez]
* efc252e 2013-06-16 | Parametrización del programa [Sergio Gómez]
* e19f2c1 2013-06-16 | Creación del proyecto [Sergio Gómez]
```

El resto de cambios no se han borrado (aún), simplemente no están accesibles porque git no sabe como referenciarlos. Si sabemos su hash podemos acceder aún a ellos. Pasado un tiempo, eventualmente Git tiene un recolector de basura que los borrará. Se puede evitar etiquetando el estado final.

De todas maneras, reset es una operación delicada, que debe evitarse, sobre todo cuando se trabaja en repositorios compartidos.

4.1.6 Modificar un commit

Esto se usa cuando hemos olvidado añadir un cambio a un commit que acabamos de realizar.

```
$ cat hola.php
<?php
// Autor: Sergio Gómez
// El nombre por defecto es Mundo
$nombre = isset($argv[1]) ? $argv[1] : "Mundo";
@print "Hola, {$nombre}\n";
$ git commit -a -m "Añadido el autor del programa"
[master cf405c1] Añadido el autor del programa
 1 file changed, 1 insertion(+)
```

El parámetro `-a` hace un `git add` antes de hacer commit de todos los archivos modificados o borrados (de los nuevos no), con lo que nos ahorramos un paso. Ahora nos percatamos que se nos ha olvidado poner el correo electrónico.

```
$ cat hola.php
<?php
// Autor: Sergio Gómez <sergio@uco.es>
// El nombre por defecto es Mundo
$nombre = isset($argv[1]) ? $argv[1] : "Mundo";
@print "Hola, {$nombre}\n";
$ git add hola.php
$ git commit --amend -m "Añadido el autor del programa y su email"
[master 96a39df] Añadido el autor del programa y su email
 1 file changed, 1 insertion(+)
$ git hist
```

```
* 96a39df 2013-06-16 | Añadido el autor del programa y su email (HEAD, ma
ster) [Sergio Gómez]
* fd4da94 2013-06-16 | Se añade un comentario al cambio del valor por def
ecto (tag: v1) [Sergio Gómez]
* 3283e0d 2013-06-16 | Se añade un parámetro por defecto (tag: v1-beta)
[Sergio Gómez]
* efc252e 2013-06-16 | Parametrización del programa [Sergio Gómez]
* e19f2c1 2013-06-16 | Creación del proyecto [Sergio Gómez]
```

4.2 Moviendo y borrando archivos

4.2.1 Mover un archivo a otro directorio con git

Para mover archivos usaremos la orden `git mv`:

```
$ mkdir lib
$ git mv hola.php lib
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   renamed:    hola.php -> lib/hola.php
#
```

4.2.2 Mover y borrar archivos.

Otra forma de hacer lo mismo:

```
$ mkdir lib
$ mv hola.php lib
$ git add lib/hola.php
$ git rm hola.php
```

Y ya podemos guardar los cambios:

```
$ git commit -m "Movido hola.php a lib."
[master 8c2a509] Movido hola.php a lib.
 1 file changed, 0 insertions(+), 0 deletions(-)
 rename hola.php => lib/hola.php (100%)
```

Esta página se ha dejado vacía a propósito

Capítulo 5

Ramas

5.1 Administración de ramas

5.1.1 Crear una nueva rama

Cuando vamos a trabajar en una nueva funcionalidad, es conveniente hacerlo en una nueva rama, para no modificar la rama principal y dejarla inestable. Aunque la orden para manejar ramas es `git branch` podemos usar también `git checkout`.

```
$ git branch hola
$ git checkout hola
Switched to branch 'hola'
```

O de forma más rápida:

```
$ git checkout -b hola
Switched to a new branch 'hola'
```

5.1.2 Modificaciones en la rama secundaria

Añadimos un nuevo archivo en el directorio `lib` llamado `HolaMundo.php`:

```
<?php
```

```
class HolaMundo
{
   private $nombre;
```

```
   function __construct($nombre)
   {
```

```
        $this->nombre = $nombre;
    }

    function __toString()
    {
        return sprintf ("Hola, %s.\n", $this->nombre);
    }
}
```

Y modificamos `hola.php`:

```
<?php
// Autor: Sergio Gómez <sergio@uco.es>
// El nombre por defecto es Mundo
require('HolaMundo.php');

$nombre = isset($argv[1]) ? $argv[1] : "Mundo";
print new HolaMundo($nombre);
```

Podríamos confirmar los cambios todos de golpe, pero lo haremos de uno en uno, con su comentario.

```
$ git add lib/HolaMundo.php
$ git commit -m "Añadida la clase HolaMundo"
[hola 6932156] Añadida la clase HolaMundo
 1 file changed, 16 insertions(+)
 create mode 100644 lib/HolaMundo.php
$ git add lib/hola.php
$ git commit -m "hola usa la clase HolaMundo"
[hola 9862f33] hola usa la clase HolaMundo
 1 file changed, 3 insertions(+), 1 deletion(-)
```

Y ahora con la orden `git checkout` podemos movernos entre ramas:

```
$ git checkout master
Switched to branch 'master'
$ git checkout hola
Switched to branch 'hola'
```

5.1.3 Modificaciones en la rama master

Podemos volver y añadir un nuevo archivo a la rama principal:

```
$ git checkout master
Switched to branch 'master'
```

```
$ cat README.md
# Curso de GIT

Este proyecto contiene el curso de introducción a GIT
$ git add README.md
$ git commit -m "Añadido README.md"
[master c3e65d0] Añadido README.md
 1 file changed, 3 insertions(+)
 create mode 100644 README.md
$ git hist --all
* c3e65d0 2013-06-16 | Añadido README.md (HEAD, master) [Sergio Gómez]
| * 9862f33 2013-06-16 | hola usa la clase HolaMundo (hola) [Sergio Góme
z]
| * 6932156 2013-06-16 | Añadida la clase HolaMundo [Sergio Gómez]
|/
* 81c6e93 2013-06-16 | Movido hola.php a lib [Sergio Gómez]
* 96a39df 2013-06-16 | Añadido el autor del programa y su email [Sergio G
ómez]
* fd4da94 2013-06-16 | Se añade un comentario al cambio del valor por def
ecto (tag: v1) [Sergio Gómez]
* 3283e0d 2013-06-16 | Se añade un parámetro por defecto (tag: v1-beta)
[Sergio Gómez]
* efc252e 2013-06-16 | Parametrización del programa [Sergio Gómez]
* e19f2c1 2013-06-16 | Creación del proyecto [Sergio Gómez]
```

Y vemos como `git hist` muestra la bifurcación en nuestro código.

5.2 Fusión de ramas y resolución de conflictos

5.2.1 Mezclar ramas

Podemos incorporar los cambios de una rama a otra con la orden `git merge`

```
$ git checkout hola
Switched to branch 'hola'
$ git merge master
Merge made by the 'recursive' strategy.
 README.md | 3 +++
 1 file changed, 3 insertions(+)
 create mode 100644 README.md
$ git hist --all
*   9c6ac06 2013-06-16 | Merge commit 'c3e65d0' into hola (HEAD, hola) [S
ergio Gómez]
|\
* | 9862f33 2013-06-16 | hola usa la clase HolaMundo [Sergio Gómez]
```

```
* | 6932156 2013-06-16 | Añadida la clase HolaMundo [Sergio Gómez]
| |
| * c3e65d0 2013-06-16 | Añadido README.md [Sergio Gómez]
|/
* 81c6e93 2013-06-16 | Movido hola.php a lib [Sergio Gómez]
* 96a39df 2013-06-16 | Añadido el autor del programa y su email [Sergio G
ómez]
* fd4da94 2013-06-16 | Se añade un comentario al cambio del valor por def
ecto (tag: v1) [Sergio Gómez]
* 3283e0d 2013-06-16 | Se añade un parámetro por defecto (tag: v1-beta)
[Sergio Gómez]
* efc252e 2013-06-16 | Parametrización del programa [Sergio Gómez]
* e19f2c1 2013-06-16 | Creación del proyecto [Sergio Gómez]
```

De esa forma se puede trabajar en una rama secundaria incorporando los cambios de la rama principal o de otra rama.

5.2.2 Resolver conflictos

Un conflicto es cuando se produce una fusión que Git no es capaz de resolver. Vamos a modificar la rama master para crear uno con la rama hola.

```
$ git checkout master
M   lib/hola.php
Already on 'master'
$ cat lib/hola.php
<?php
// Autor: Sergio Gómez <sergio@uco.es>
print "Introduce tu nombre:";
$nombre = trim(fgets(STDIN));
@print "Hola, {$nombre}\n";
$ git add lib/hola.php
$ git commit -m "Programa interactivo"
[master 9c85275] Programa interactivo
 1 file changed, 2 insertions(+), 2 deletions(-)
$ git hist --all
*   9c6ac06 2013-06-16 | Merge commit 'c3e65d0' into hola (hola) [Sergio
Gómez]
|\
* | 9862f33 2013-06-16 | hola usa la clase HolaMundo [Sergio Gómez]
* | 6932156 2013-06-16 | Añadida la clase HolaMundo [Sergio Gómez]
| | * 9c85275 2013-06-16 | Programa interactivo (HEAD, master) [Sergio Gó
mez]
| |/
| * c3e65d0 2013-06-16 | Añadido README.md [Sergio Gómez]
```

```
|/
* 81c6e93 2013-06-16 | Movido hola.php a lib [Sergio Gómez]
* 96a39df 2013-06-16 | Añadido el autor del programa y su email [Sergio G
ómez]
* fd4da94 2013-06-16 | Se añade un comentario al cambio del valor por def
ecto (tag: v1) [Sergio Gómez]
* 3283e0d 2013-06-16 | Se añade un parámetro por defecto (tag: v1-beta)
[Sergio Gómez]
* efc252e 2013-06-16 | Parametrización del programa [Sergio Gómez]
* e19f2c1 2013-06-16 | Creación del proyecto [Sergio Gómez]
```

Volvemos a la rama hola y fusionamos:

```
$ git checkout hola
Switched to branch 'hola'
$ git merge master
Auto-merging lib/hola.php
CONFLICT (content): Merge conflict in lib/hola.php
Automatic merge failed; fix conflicts and then commit the result.
$ cat lib/hola.php
<?php
// Autor: Sergio Gómez <sergio@uco.es>
<<<<<<< HEAD
// El nombre por defecto es Mundo
require('HolaMundo.php');
```

```
$nombre = isset($argv[1]) ? $argv[1] : "Mundo";
# print new HolaMundo($nombre);print "Introduce tu nombre:";
$nombre = trim(fgets(STDIN));
@print "Hola, {$nombre}\n";
>>>>>>> master
```

La primera parte marca el código que estaba en la rama donde trabajábamos (HEAD) y la parte final el código de donde fusionábamos. Resolvemos el conflicto:

```
$ cat lib/hola.php
<?php
// Autor: Sergio Gómez <sergio@uco.es>
require('HolaMundo.php');
```

```
print "Introduce tu nombre:";
$nombre = trim(fgets(STDIN));
print new HolaMundo($nombre);
$ git add lib/hola.php
```

```
$ git commit -m "Solucionado el conflicto al fusionar con la rama master"
[hola a36af04] Solucionado el conflicto al fusionar con la rama master
```

5.2.3 Rebasing vs Merging

Rebasing es otra técnica para fusionar distinta a merge y usa la orden `git rebase`. Vamos a dejar nuestro proyecto como estaba antes del fusionado:

```
$ git checkout hola
Switched to branch 'hola'
$ git hist
*   a36af04 2013-06-16 | Solucionado el conflicto al fusionar con la ram
a master (HEAD, hola) [Sergio Gómez]
|\
| * 9c85275 2013-06-16 | Programa interactivo (master) [Sergio Gómez]
* |   9c6ac06 2013-06-16 | Merge commit 'c3e65d0' into hola [Sergio Góme
z]
|\ \
| |/
| * c3e65d0 2013-06-16 | Añadido README.md [Sergio Gómez]
* | 9862f33 2013-06-16 | hola usa la clase HolaMundo [Sergio Gómez]
* | 6932156 2013-06-16 | Añadida la clase HolaMundo [Sergio Gómez]
|/
* 81c6e93 2013-06-16 | Movido hola.php a lib [Sergio Gómez]
* 96a39df 2013-06-16 | Añadido el autor del programa y su email [Sergio G
ómez]
* fd4da94 2013-06-16 | Se añade un comentario al cambio del valor por def
ecto (tag: v1) [Sergio Gómez]
* 3283e0d 2013-06-16 | Se añade un parámetro por defecto (tag: v1-beta)
[Sergio Gómez]
* efc252e 2013-06-16 | Parametrización del programa [Sergio Gómez]
* e19f2c1 2013-06-16 | Creación del proyecto [Sergio Gómez]
$ git reset --hard 9862f33
HEAD is now at 9862f33 hola usa la clase HolaMundo
```

```
$ git checkout master
Switched to branch 'master'
$ git hist
* 9c85275 2013-06-16 | Programa interactivo (HEAD, master) [Sergio Gómez]
* c3e65d0 2013-06-16 | Añadido README.md [Sergio Gómez]
* 81c6e93 2013-06-16 | Movido hola.php a lib [Sergio Gómez]
* 96a39df 2013-06-16 | Añadido el autor del programa y su email [Sergio G
ómez]
* fd4da94 2013-06-16 | Se añade un comentario al cambio del valor por def
ecto (tag: v1) [Sergio Gómez]
```

```
* 3283e0d 2013-06-16 | Se añade un parámetro por defecto (tag: v1-beta)
[Sergio Gómez]
* efc252e 2013-06-16 | Parametrización del programa [Sergio Gómez]
* e19f2c1 2013-06-16 | Creación del proyecto [Sergio Gómez]
$ git reset --hard c3e65d0
HEAD is now at c3e65d0 Añadido README.md
```

Y nuestro estado será:

```
$ git hist --all
* 9862f33 2013-06-16 | hola usa la clase HolaMundo (hola) [Sergio Gómez]
* 6932156 2013-06-16 | Añadida la clase HolaMundo [Sergio Gómez]
| * c3e65d0 2013-06-16 | Añadido README.md (HEAD, master) [Sergio Gómez]
|/
* 81c6e93 2013-06-16 | Movido hola.php a lib [Sergio Gómez]
* 96a39df 2013-06-16 | Añadido el autor del programa y su email [Sergio G
ómez]
* fd4da94 2013-06-16 | Se añade un comentario al cambio del valor por def
ecto (tag: v1) [Sergio Gómez]
* 3283e0d 2013-06-16 | Se añade un parámetro por defecto (tag: v1-beta)
[Sergio Gómez]
* efc252e 2013-06-16 | Parametrización del programa [Sergio Gómez]
* e19f2c1 2013-06-16 | Creación del proyecto [Sergio Gómez]
```

```
$ git checkout hola
Switched to branch 'hola'
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: Añadida la clase HolaMundo
Applying: hola usa la clase HolaMundo
$ git hist --all
* 491f1d2 2013-06-16 | hola usa la clase HolaMundo (HEAD, hola) [Sergio G
ómez]
* c13d75c 2013-06-16 | Añadida la clase HolaMundo [Sergio Gómez]
* c3e65d0 2013-06-16 | Añadido README.md (master) [Sergio Gómez]
* 81c6e93 2013-06-16 | Movido hola.php a lib [Sergio Gómez]
* 96a39df 2013-06-16 | Añadido el autor del programa y su email [Sergio G
ómez]
* fd4da94 2013-06-16 | Se añade un comentario al cambio del valor por def
ecto (tag: v1) [Sergio Gómez]
* 3283e0d 2013-06-16 | Se añade un parámetro por defecto (tag: v1-beta)
[Sergio Gómez]
* efc252e 2013-06-16 | Parametrización del programa [Sergio Gómez]
* e19f2c1 2013-06-16 | Creación del proyecto [Sergio Gómez]
```

Lo que hace rebase es volver a aplicar todos los cambios a la rama máster, desde su nodo más reciente. Eso significa que se modifica el orden o la historia de creación de los cambios. Por eso rebase no debe usarse si el orden es importante o si la rama es compartida.

5.3 Mezclando con la rama master

Ya hemos terminado de implementar los cambios en nuestra rama secundaria y es hora de llevar los cambios a la rama principal. Usamos `git merge` para hacer una fusión normal:

```
$ git checkout master
Switched to branch 'master'
$ git merge hola
Updating c3e65d0..491f1d2
Fast-forward
 lib/HolaMundo.php | 16 ++++++++++++++++
 lib/hola.php      |  4 +++-
 2 files changed, 19 insertions(+), 1 deletion(-)
 create mode 100644 lib/HolaMundo.php
```

Vemos que indica que el tipo de fusión es *fast-forward*. Este tipo de fusión tiene el problema que no deja rastro de la fusión, por eso suele ser recomendable usar el parámetro --no-ff para que quede constancia siempre de que se ha fusionado una rama con otra.

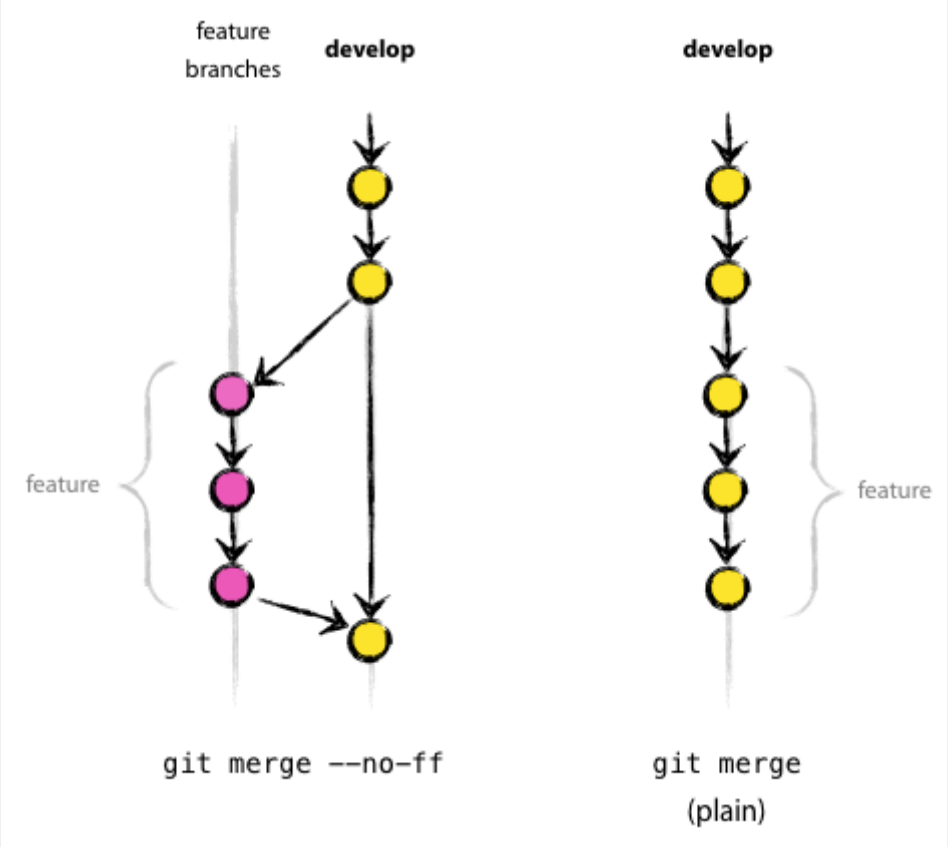

Figura 5.1 Diferencias entre tipos de fusión

Esta página se ha dejado vacía a propósito

Capítulo 6

Flujo de trabajo con Git (git flow)

6.1 La importancia de la organización del flujo de trabajo

En la introducción vimos los diferentes esquemas de organización externa de los repositorios (es decir, en lo relativo a los usuarios que componen el equipo de trabajo). Pero el repositorio en sí también tiene su esquema de organización. En los ejemplos hemos visto que usabamos una rama máster y creábamos ramas para añadir funcionalidades que luego integrábamos. Es un forma de trabajar de las muchas que hay propuestas, posiblemente la más simple, pero tiene el inconveniente de dejar la rama máster a expensas de una mala actualización y quedarnos sin una rama estable. Por eso, hay otras propuestas mejores que permiten separar el trabajo de desarrollo con el mantenimiento de las versiones estables. Una de las más conocidas es la propuesta por Vincent Driessen (http://nvie.com/posts/a-successful-git-branching-model/) y que podemos ver en la figura siguiente.

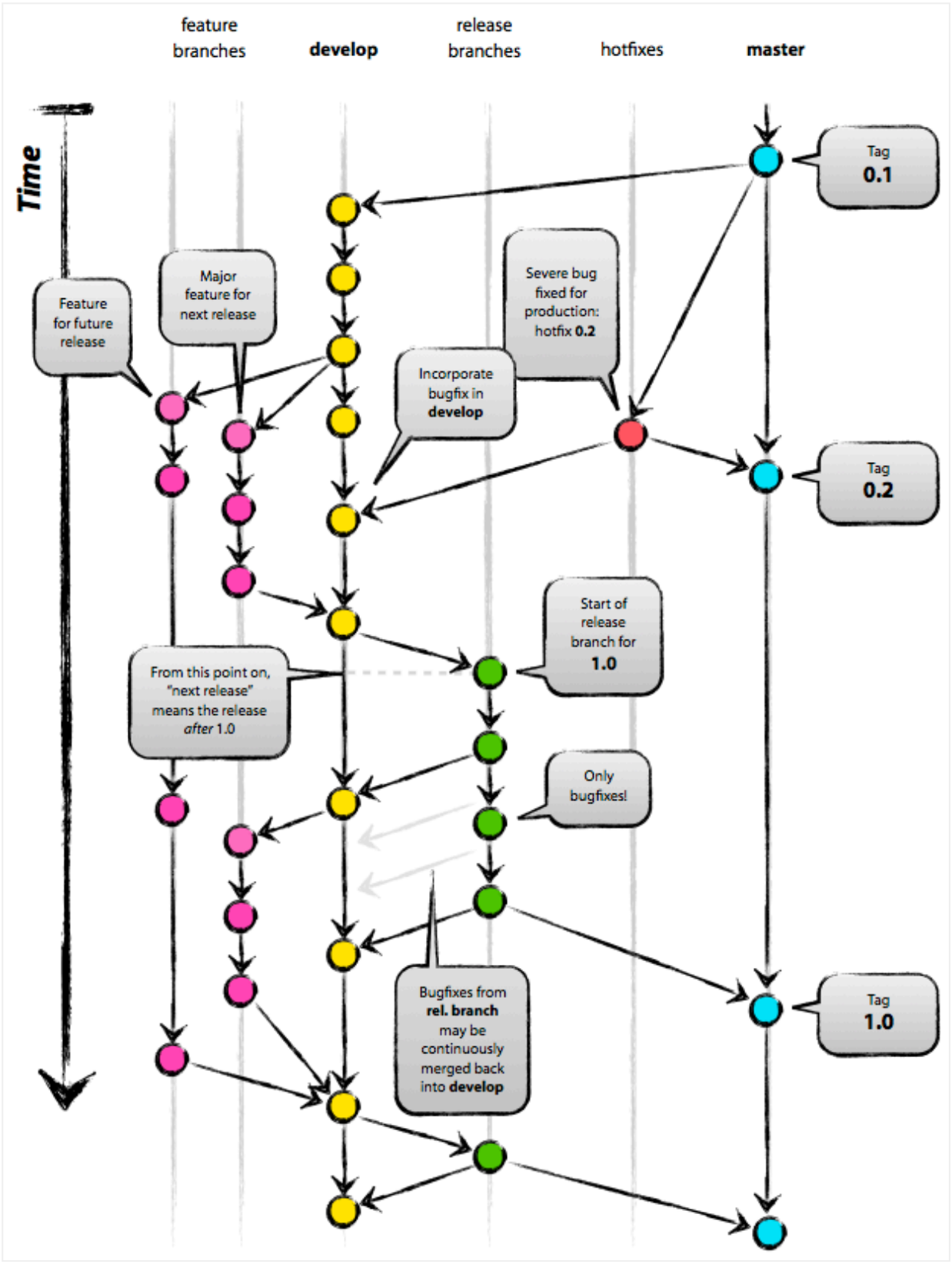

Figura 6.1 Git flow

6.1.1 Las ramas principales

En este esquema hay dos ramas principales con un tiempo de vida indefinido:

- master (*origin/master*): el código apuntado por $HEAD$ siempre contiene un estado listo para producción.
- develop (*origin/develop*): el código apuntado por $HEAD$ siempre contiene los últimos cambios desarrollados para la próxima versión del software. También se le puede llamar *rama de integración*. No es necesariamente estable.

Cuando el código de la rama de desarrollo es lo suficientemente estable, se integra con la rama master y una nueva versión es lanzada.

6.1.2 Las ramas auxiliares

Para labores concretas, pueden usarse otro tipo de ramas, las cuales tienen un tiempo de vida definido. Es decir, cuando ya no son necesarias se eliminan:

- Ramas de funcionalidad (feature branches)
- Ramas de versión (release branches)
- Ramas de parches (hotfix branches)

6.1.2.1 Feature branches

- Pueden partir de: develop
- Deben fusionarse con: develop
- Convenición de nombres: feature-NUMissue-*.

6.1.2.2 Release branches

- Pueden partir de: develop
- Deben fusionarse con: develop y master
- Convenición de nombres: release-*

6.1.2.3 Hotfix branches

- Pueden partir de: master
- Deben fusionarse con: develop y master
- Convenición de nombres: hotfix-*

6.2 La extensión flow de Git

Una de las ventajas de Git es que, además, es extensible. Es decir, se pueden crear nuevas órdenes como si de plugins se tratara. Una de las más usadas es gitflow

(https://github.com/nvie/gitflow) , que está basada en el artículo que hablamos al principio de este capítulo.

6.2.1 Instalación

Aunque la fuente original de la extensión es del mismo autor del artículo, el código no se encuentra ya muy actualizado y hay un fork bastante más activo en petervanderdoes/gitflow (https://github.com/petervanderdoes/gitflow) . En el wiki del repositorio están las instrucciones de instalación (https://github.com/petervanderdoes/gitflow/wiki) para distintos sistemas. Una vez instalados tendremos una nueva órden: `git flow`.

En cualquier caso, en Ubuntu/Debian podemos instalarlo a través de apt-get:

```
$ apt-get install git-flow
```

Y en OSX con `homebrew` si ya estamos usando su versión de git:

```
$ brew install git-flow
```

6.2.2 Uso

Para cambiar a las ramas master y develop, seguiremos usando `git checkout`, pero para trabajar con las ramas antes indicadas gitflow nos facilita las siguientes órdenes:

- `git flow init`: Permite inicializar el espacio de trabajo.
- `git flow feature`: Permite crear y trabajar con ramas de funcionalidades.
- `git flow release`: Permite crear y trabajar con ramas de versiones.
- `git flow hotfix`: Permite crear y trabajar con ramas de parches.

Licencia

Figura .1 CC-BY-SA

Las imágenes pertenecen al libro oficial de Git (https://git-scm.com/doc).

Referencias

- Documentación oficial en inglés (http://git-scm.com/documentation).
- Documentación oficial en español (quizás incompleta) (http://git-scm.com/book/es).
- Curso de Git (inglés) (http://gitimmersion.com/lab_01.html). La mayoría de la documentación de este manual está basada en este curso.
- Curso interactivo de Git (inglés) (http://try.github.io/levels/1/challenges/1).
- Página de referencia de todas las órdenes de Git (inglés) (http://gitref.org/).
- Chuleta con las órdenes más usuales de Git (http://byte.kde.org/~zrusin/git/git-cheat-sheet-large.png).
- Gitmagic (ingles y español). Otro manual de Git (http://www-cs-students.stanford.edu/~blynn/gitmagic/intl/es/)
- Artículo técnico: Un modelo exitoso de ramificación en Git (http://nvie.com/posts/a-successful-git-branching-model/).