Sistemas de Control de Versiones

1.3 Git avanzado

Elena García-Morato, Felipe Ortega, Enrique Soriano, Gorka Guardiola

GSyC, ETSIT. URJC.

Grupo de Innovación Docente Laboratorio de Ciencia de Datos para la Innovación de la Enseñanza (DSLAB-TI)

6 de julio, 2023

DSLAB-TI
DATA SCIENCE LABORATORY
FOR TEACHING INNOVATION

Universidad
Rey Juan Carlos

Sistemas de Control de Versiones

1.3 Git avanzado

Contenidos

1.3.1 Ramas

Ciclo de cambios en archivos del proyecto

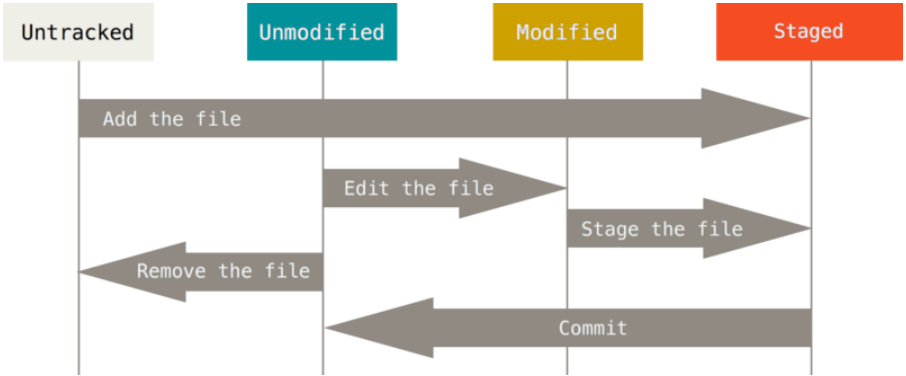

Figura 1: Resumen del ciclo de registro de cambios en los archivos de un proyecto Git. Fuente: git-scm (Sec. 2.2).

Propósito de las ramas

- Si todo el mundo trabaja simultáneamente en la misma copia del proyecto será muy habitual que surjan problemas: conflictos por cambios incompatibles al hacer un *merge*, etc.
- Casi todos los SCV incluyen soporte para crear ramas (*branches*).
- Una rama es, a todos los efectos, una versión alternativa del proyecto. En concreto, en Git se crean a partir de una instantánea del estado actual del proyecto.
- Hábito de trabajo: cuando creamos una nueva *funcionalidad* o arreglamos un *problema* primero **creamos una nueva rama** para trabajar dentro de ella.
  - Los cambios en esa rama no son visibles en el resto hasta que la volvamos a integrar en otra rama o en la rama principal.
  - Así se evita filtrar código inestable en la rama principal del proyecto. También permite organizar y limpiar todos los cambios antes de integrarlos de vuelta.

Ramas en un proyecto

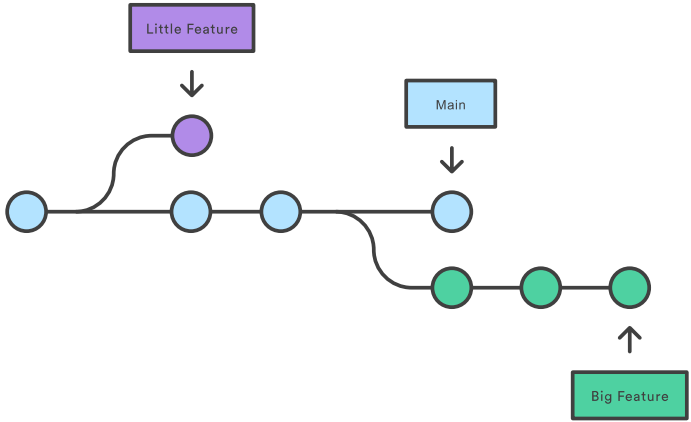

Figura 2: Esquema que ilustra la utilidad de la creación de ramas en un repo Git. Fuente: Tutorial de Atlassian Bitbucket.

Creación de ramas

- Crear nuevas ramas en Git es un proceso rápido (casi instantáneo) y no consume grandes recursos.
- Cambiar de una rama a otra es igual de fácil y rápido. Por eso, se incentiva la creación de ramas (una para cada bug, para cada nueva feature, etc.).
- Recordemos como se construye el grafo de cambios.
  - Cada *commit* contiene un puntero a una instantánea del los cambios añadidos al *stage*, nombre y e-mail del autor, mensaje descriptivo, y un puntero o varios a los ancestros.
  - El *commit* inicial no tiene ancestros, un *commit* normal tiene un ancestro y un *merge* tiene dos o más.

Creación de ramas

- Supongamos un nuevo repo en el que añadimos tres ficheros.

```
$ git init
$ git add README test.rb LICENSE
$ git commit -m "Primer commit"
```

Creación de ramas

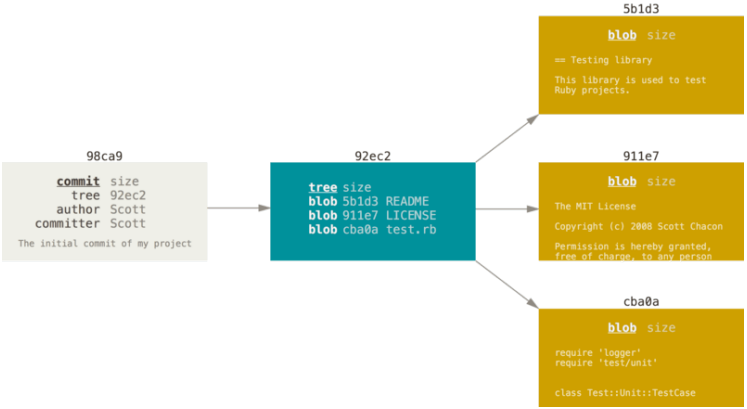

Figura 3: Commit inicial y su árbol. Fuente: Pro Git.

Creación de ramas

- Cuando hacemos dos cambios más, cada cambio guarda una instantánea del estado del proyecto.

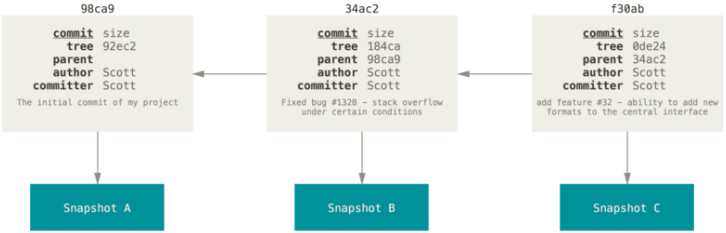

Figura 4: Grafo de cambios con tres *commits*. Fuente: Pro Git.

Creación de ramas

- Una rama es un puntero a uno de estos *commits* y su instantánea del proyecto. Por defecto, al crear un repo se crea una rama `main` o `master`.
- Al hacer *commits*, el puntero se mueve adelante automáticamente.

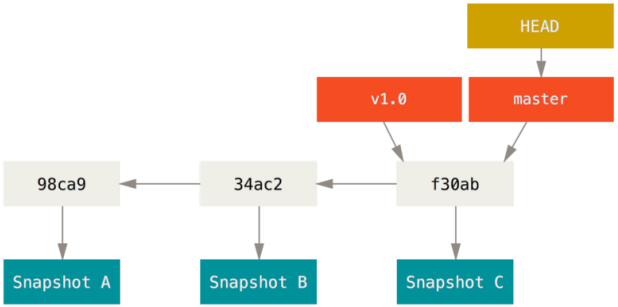

Figura 5: La rama `master` apunta al último *commit*. Aquí, también se ha etiquetado como `v1.0`. Fuente: Pro Git.

Creación de ramas

- Al crear una nueva rama surge otro puntero que apunta al mismo *commit* que la rama actual.

```
$ git branch testing
```

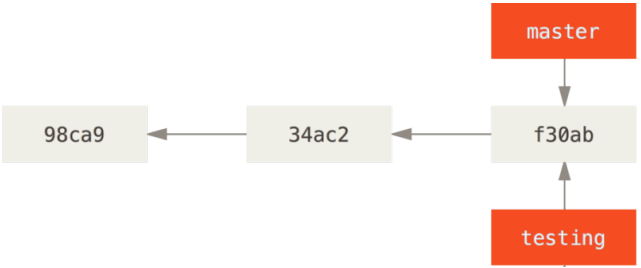

Figura 6: Se crea la rama `testing`, que también apunta al último *commit*. Fuente: Pro Git.

Creación de ramas

- El puntero HEAD es el que indica mi rama actual de trabajo.

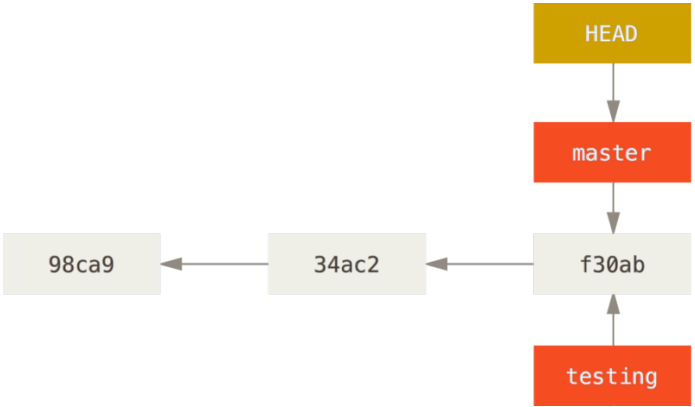

Figura 7: El puntero HEAD apunta a la rama `master`. Fuente: Pro Git.

Creación de ramas

- Vemos que las dos ramas están apuntando al mismo *commit*.

```
$ git log --oneline --decorate
f30ab (HEAD -> master, testing) Add feature #32 - ability to add new formats to central intf
34ac2 Fix bug #1328 - stack overflow under certain conditions
98ca9 Initial commit
```

Cambio de rama

- Para cambiar de rama uso el comando `git checkout branch-name`.

```
$ git checkout testing
```

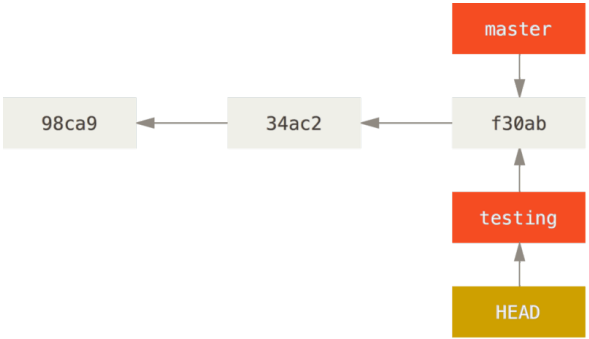

Figura 8: El puntero `HEAD` ahora apunta a la rama `testing`. Fuente: Pro Git.

Commits en la nueva rama

```
$ vim test.rb
$ git commit -a -m 'made a change'
```

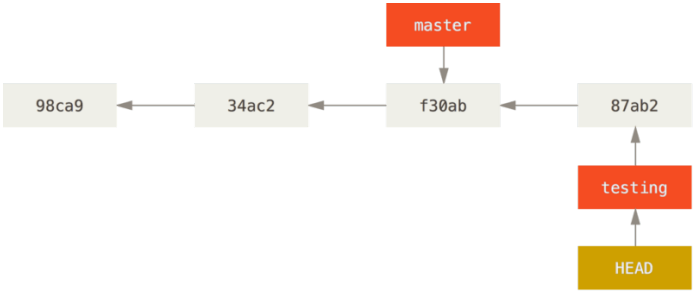

Figura 9: El puntero `HEAD` y la rama `testing` reflejan el último cambio. La rama `main` no avanza. Fuente: Pro Git.

Commits en la nueva rama

- Ahora, volvemos a la rama `master` y hacemos un *commit*.

```
$ git checkout master
$ vim test.rb
$ git commit -a -m 'made other changes'
```

- Ahora tenemos una divergencia en el historial de cambios del proyecto.

Commits en la nueva rama

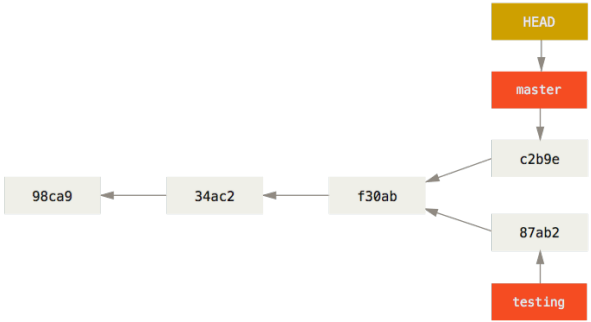

Figura 10: Nuevo historial divergente de cambios del proyecto. Fuente: Pro Git.

Mostrar el historial de cambios

- Cuidado con el comando `git log`. Por defecto, solo nos muestra el historial de cambios de **la rama actualmente en uso**.
- Para mostrar los cambios de una rama en concreto: `git log nombre-rama`.
- Para mostrar los cambios de todas las ramas: `git log -all`.

```
$ 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 central interface
* 34ac2 Fix bug #1328 - stack overflow under certain conditions
* 98ca9 initial commit of my project
```

Comando `git switch`

- Desde Git v2.23 se puede usar el comando `git switch`.

```
$ git switch testing  # Cambiar a la rama testing
$ git switch -c new-branch  # Crea la nueva rama new-branch
$ git switch - # Cambia a la rama en la que estabas previamente
```

Tutoriales sobre ramas

- Ejemplo paso a paso de creación e integración de ramas (*merging*).
- Gestión de ramas.
  - Mucho cuidado con renombrar la rama principal (`main` o `master`). Podemos romper muchas utilidades y servicios integrados con nuestro repositorio.
- *Stashing* y limpieza de cambios.
  - Si tenemos cambios pendientes de *commit* en una rama Git no nos deja cambiar a otra. Primero hay que "salvar" los cambios (stash).
- Estrategias de trabajo con ramas.

1.3.2 Deshacer cambios

Modificación de *commits*

- Un caso habitual es que olvidamos añadir uno o varios archivos a un cambio y hacemos *commit*
- El fallo se puede arreglar fácilmente con el comando `git commit --amend`.
- Como resultado, el *commit* inicial se descarta y se reemplaza por completo con la nueva versión de cambios, como si el anterior nunca hubiese ocurrido.

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

- Precaución: Solo se debe hacer un `git commit --ammend` **sobre cambios que no se han enviado con `git push`** a otro repo remoto. Si lo hacemos y forzamos los cambios en esa rama podemos crear problemas a otros colaboradores.

Gestión del área *staging*

- Otro caso habitual es descartar los últimos cambios que hemos añadido al área de *stating*.

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

renamed:    README.md -> README
modified:   CONTRIBUTING.md
```

- Se puede observar que la propia salida del comando nos sugiere la fórmula para deshacer la operación: `(use "git reset HEAD <file>..." to unstage)`.

Gestión del área *staging*

```
$ git reset HEAD CONTRIBUTING.md
Unstaged changes after reset:
M^^ICONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
(use "git reset HEAD <file>..." to unstage)

renamed:    README.md -> README

Changes not staged for commit:
(use "git add <file>..." to update what will be committed)
(use "git checkout -- <file>..." to discard changes in working directory)

modified:   CONTRIBUTING.md
```

Gestión del área de trabajo

- Si queremos descartar por completo los últimos cambios en un archivo antes de añadirlo al área de *staging* usamos:

```
$ git status
Changes not staged for commit:
(use "git add <file>..." to update what will be committed)
(use "git checkout -- <file>..." to discard changes in working directory)

modified:   CONTRIBUTING.md
$ git checkout -- CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
(use "git reset HEAD <file>..." to unstage)

renamed:    README.md -> README
```

Gestión del área de trabajo

- Precaución: Mucho cuidado con el comando `git checkout -- <file>`. **La operación no se puede revertir**.
- Git no tiene "papelera de reciclaje" ni ninguna salvaguarda similar.
- La versión modificada del archivo se sobrescribe con la última versión almacenada (en el último *commit* realizado). Debemos estar absolutamente seguros de que es eso lo que queremos.

Comando `git restore`

- A partir de Git v2.23.0 se puede usar el comando git restore para realizar cambios equivalentes en el área de *staging* o el área de trabajo.

```
git restore --staged <file>  # Para deshacer cambios en staging area
git restore <file> # Para descartar cambios en un archivo del dir de trabajo
```

- En ese caso, los mensajes de sugerencia de `git status` ya aparecen actualizados.

1.3.3 Pull requests

Cómo contribuir a un proyecto

- Existen diversas maneras de contribuir a un proyecto software que está bajo control con Git.
- Formas de contribuir a un proyecto con Git.
- Aquí nos centramos en un caso común: la creación e incorporación de un `pull request`.
- Existe soporte específico en servicios como GitHub o GitLab para este caso.
  - GitHub: Crear pull request a partir de un fork e integrar un pull request.
  - GitLab: Crear *merge requests*.

Propósito de un *pull request*

- Queremos enviar cambios a un repositorio en el que no tenemos privilegios de edición.
- El primer paso es clonar el repo principal y crear una nueva rama para implementar nuestros cambios en ella.

```
$ git clone <url>
$ cd project
$ git checkout -b featureA
... work ...
$ git commit
... work ...
$ git commit
```

Realizar *pull request*

- Ahora, en el servicio de gestión de proyectos (GitHub, GitLab, etc.) tenemos que hacer un *fork* del proyecto principal para crear una copia que cuelgue de nuestra cuenta de usuario/a.
- Anotamos la URL del *fork* que hemos creado y lo usamos:

```
$ git remote add myfork <url>
$ git push -u myfork featureA
```

- Por último seguimos los pasos en la plataforma de gestión para crear el `pull request` de forma interactiva (instrucciones para GitHub).
- El comando `git request-pull` genera información sobre la petición que se puede enviar a los gestores del proyecto mediante un correo-e de contacto.

1.3.4 Etiquetado

Propósito de las etiquetas

- Como en otros SCV, Git permite poner etiquetas a puntos específicos en el historial del repositorio para marcar hitos relevantes.
- Normalmente se suele usar para versionado (v1.0, v1.1, etc.).
- El comando que muestra todas las etiquetas de un repo es `git tag`.
- El uso de la opción `-l` o `--list` es obligatorio si usamos comodines para patrones de búsqueda.

```
$ git tag
v1.0
v2.0
$ git tag -l "v1.8.5*" # Busca etiquetas que coinciden con un patrón
v1.8.5
v1.8.5-rc0
v1.8.5-rc1
v1.8.5.1
```

Creación de etiquetas

- Se pueden crear etiquetas de dos tipos: `lightweight` o `annotated`.
- Una etiqueta `lightweight` es como una rama sin recorrido (un puntero a una instantánea del proyecto). Sirven para marcar hitos internos o de poca importancia.
- Una etiqueta `annotated` se usa para hitos relevantes, como versiones publicadas. Son objetos completos, guardados en la base de datos del repo Git, que incluyen:
  - *Checksum*.
  - Nombre y correo-e del etiquetador.
  - Fecha.
  - Mensaje de descripción.
  - Se pueden firmar de forma segura (por ejemplo con GPG).

Creación de etiquetas

- Para crear etiquetas annotated:

```
$ git tag -a v1.4 -m "my version 1.4"
$ git tag
v0.1
v1.3
v1.4
```

Creación de etiquetas

```
$ git show v1.4
tag v1.4
Tagger: Supercoco Coder <super@coco.cc>
Date:   Fri May 6 21:19:12 2022 +0100

my version 1.4

commit ca82a6dff817ec66f44342007202690a93763949
Author: Count Counting <count@sesame-street.com>
Date:   Mon Mar 21 21:52:11 2022 -0700

Change version number
```

Creación de etiquetas

- Para crear etiquetas `lightweight`:

```
$ git tag v1.4-lw
$ git tag
...
$ git show v1.4-lw
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

Change version number
```

Creación de etiquetas

- Para anotar commits anteriores al HEAD:

```
$ git log --pretty=oneline
15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'
a6b4c97498bd301d84096da251c98a07c7723e65 Create write support
0d52aaab4479697da7686c15f77a3d64d9165190 One more thing

$ git tag -a v1.2 0d52aaa
```

Compartir etiquetas

- Por defecto, las etiquetas no se envían a los repos remotos en una operación `git push`.
- Se deben enviar explícitamente usando ese mismo comando:

```
$ git push origin v1.5
Counting objects: 14, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (12/12), done.
Writing objects: 100% (14/14), 2.05 KiB | 0 bytes/s, done.
Total 14 (delta 3), reused 0 (delta 0)
To git@github.com:schacon/simplegit.git
* [new tag]         v1.5 -> v1.5
```

Compartir etiquetas

- Si hay muchas etiquetas pendientes de enviar al repo remoto, se puede usar el comando `git push origin –tags`.
- Transfiere al repo remoto todas las etiquetas locales, tanto `lightweight` como `annotated` que todavía no estén allí.

```
$ git push origin --tags
Counting objects: 1, done.
Writing objects: 100% (1/1), 160 bytes | 0 bytes/s, done.
Total 1 (delta 0), reused 0 (delta 0)
To git@github.com:schacon/simplegit.git
* [new tag]         v1.4 -> v1.4
* [new tag]         v1.4-lw -> v1.4-lw
```

- Para enviar solo las etiquetas `annotated` se usa `git push <remote> –follow-tags`.

Borrar etiquetas

- Para borrar etiquetas del repo local, se usa `git tag -d <tagname>`.
- Para borrar etiquetas del repo remoto hay dos alternativas:

```
$ git push <remote> :refs/tags/<tagname>
$ git push origin --delete <tagname>
```

- Para enviar solo las etiquetas annotated se usa `git push <remote> -follow-tags`.

Lecturas sugeridas

Para saber más

- La referencia fundamental sobre Git accesible de forma pública es el libro Pro Git, disponible en inglés y castellano [1].
- Los libros de la serie *Head First* de O'Reilly son muy conocidos por su nivel muy accesible y su enfoque didáctico. Para este tema, se ha publicado en enero de 2022 *Head First Git* [2].
- Otra referencia muy conocida de la misma editorial es [3].

Referencias I

[1] S. Chacon y B. Straub. *Pro Git*. The Expert's Voice. Apress, 2014.

[2] Raju Gandhi. *Head First Git. A Learner's Guide to Understand Git from the Inside Out*. Head First. O'Reilly Media, 2022.

[3] P.K. Ponuthorai y J. Loeliger. *Version Control with Git*. 3ª ed. O'Reilly Media, 2022.